SEEING SPARSITY: PATTERN-GUIDED AGENTS FOR HARDWARE-ADAPTIVE SPMV OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Sparse matrix-vector multiplication (SpMV) performance depends on both matrix sparsity and GPU architecture, making it difficult to choose an effective kernel from structure alone. We present SPARROW, which combines a sparsity-aware decision model with feedback-guided kernel search. The model encodes local nonzero occupancy patches with a Transformer and global matrix statistics with an MLP to rank expert-defined strategies for work mapping and reduction. An agent starts from these strategies, tunes template parameters, and generates or revises kernel code using compilation, correctness checks, target-GPU measurements, and available profiler feedback in a search tree. Within its search budget, it returns the lowest-latency kernel that passes its correctness checks. We evaluate SPARROW on NVIDIA A100 and Hygon DCU GPUs against vendor libraries, a supervised strategy selector, and general-purpose code agents. With GLM, SPARROW achieves a $1.74\times$ speedup over cuSparse on NVIDIA A100 and a $3.9\times$ speedup over hipSparse on Hygon DCU. In ablations, throughput falls from 207.38 GFLOPS to 190.33 without sparsity awareness and 134.59 without the agent search. The results show that structure-based strategy selection and target-device feedback both contribute to SpMV kernel performance across the evaluated GPUs.

1 INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a fundamental operation in scientific computing, graph analytics, and iterative solvers, and its efficiency directly affects the runtime of many large-scale applications Bell & Garland (2009); Ashari et al. (2014); Saad (2003). Although the computation is simple, its performance depends strongly on the input matrix and hardware architecture Bell & Garland (2009); Steinberger et al. (2017). Two matrices with identical dimensions and nonzero counts can behave quite differently. One may concentrate its nonzeros in a few long rows, while the other distributes them evenly across many short rows. The former can cause thread load imbalance, while the latter may offer limited opportunities for cooperation among threads Merrill & Garland (2016b); Steinberger et al. (2017). The relative performance of the same tiling and reduction strategies can also vary across GPUs with different cache capacities, memory bandwidths, and execution resources. Optimizing SpMV therefore requires understanding diverse sparsity structures and mapping them to execution strategies suited to the target hardware Sedaghati et al. (2015); Du et al. (2022).

Prior work has approached this problem through three overlapping lines of research. The first relies on expert-designed **storage formats** and kernels, using tiling, load balancing, and parallel reduction to deliver robust performance across a broad range of inputs Bell & Garland (2009); Merrill & Garland (2016b); Steinberger et al. (2017). These methods draw primarily on domain knowledge encoded in their implementations. The second uses **adaptive search** to explore storage layouts, computational strategies, and parameter combinations for individual matrices Sedaghati et al. (2015); Du et al. (2022). For example, AlphaSparse organizes expert optimization operations into an operator graph and uses search and pruning to generate matrix-specific storage formats and SpMV implementations Du et al. (2022). The third uses **learning model** to acquire code generation and optimization capabilities. SparseRL encodes the positions of matrix nonzeros as model inputs and trains a model to generate CUDA kernels through reinforcement learning that accounts for both correctness and execution efficiency Wang et al. (2026). Kevin incorporates generation, execution,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

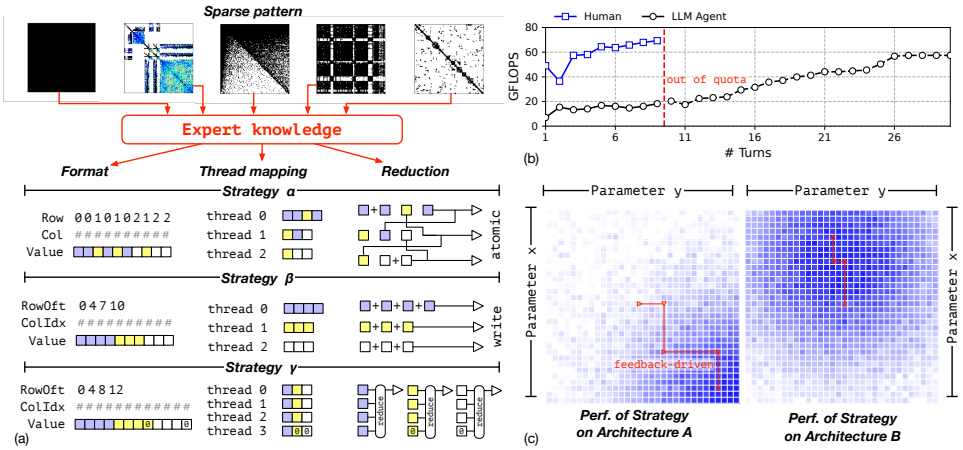


Figure 1: Human experts design kernel for sparse problem based on sparsity pattern and then refine them on specific architectures.

and feedback-based revision into multi-turn reinforcement learning, teaching a model to improve kernels over successive iterations Baronio et al. (2026). Together, these approaches show how expert knowledge, learned decisions, and execution feedback can inform automated kernel optimization.

Existing methods still face trade-offs among generation efficiency, kernel performance, and generalization. Handcrafted kernels are readily available, but their finite set of strategies cannot cover every combination of sparsity structure and hardware. Broader search spaces may yield better implementations, at the cost of additional compilation and measurement. AlphaSparse, for instance, can construct high-performance implementations for diverse matrices, but its search takes hours. Learning methods move some optimization effort into training, yet they cannot readily adapt to the vast diversity of sparsity patterns and rely on costly environment feedback for further improvement through online learning. Together, these limitations create a practical trilemma in domain-specific kernel optimization, making it difficult to achieve fast generation, high kernel performance, and broad generalization simultaneously.

Our approach follows the intuition of human experts. Figure 1 (a) shows how they match sparsity patterns to storage formats, thread mappings, and reductions. Block clusters suggest tiling and reuse, while long rows suggest work partitioning. This learned association is an inductive bias that a general-purpose LLM agent lacks. As the Figure 1 (b) illustrates, discovering it may require lengthy in-context learning and many rounds of feedback. Yet intuition provides only a promising direction. The Figure 1 (c) shows that the best configurations shift across hardware architectures, so feedback from the target GPU is needed to refine the choice. A good starting point can substantially shorten the search for a high-performing kernel. This suggests a **two-stage process** that uses decision model to choose a promising strategy before testing and refining it through hardware measurements. These stages echo the fast intuition and deliberate reasoning associated with System 1 and System 2 in human cognition, respectively Kahneman (2003). A dedicated decision model learns the pattern-to-strategy association and selects an initial expert strategy. An agent then tests and revises it on the target hardware.

We propose SPARROW, an SpMV kernel generation system that pairs a learned prior over expert strategies with search on the target GPU. It uses matrix structure to choose a starting strategy, then generates and refines kernels using execution feedback. This work makes three contributions.

- We represent sparsity at two scales. A Transformer encodes local occupancy-map patches, while an MLP encodes global statistics such as matrix dimensions, nonzero count, and density. Their combined representation captures structural patterns for strategy selection.
- We learn a matrix-to-strategy prior over expert-defined SpMV templates, including choices about tiling, thread mapping, and reduction. The decision model selects a structure-matched starting point for the agent before measurements on the target GPU.

- We use feedback-guided tree search to turn that strategy into kernel code and maintain alternative candidates. The agent checks correctness, measures GFLOPS and available profiler metrics for memory access and occupancy, then tunes parameters such as tile size, switches strategies, or revises code backbones. Within its search budget, it returns the lowest-latency candidate that passes validation on the target GPU.

2 Related Work

2.1 SpMV Challegen: Data and Architecture Diversity

SpMV performance changes with both the sparse pattern and the GPU: row lengths determine load imbalance, nonzero locality affects memory traffic, and the available memory and compute units determine which mapping is effective. In HPC community, prior work therefore specializes storage and execution to recurring patterns. Storage formats and thread mappings were studied in early GPU SpMV implementations Bell & Garland (2009), graph matrices motivated specialized kernels Ashari et al. (2014), while uneven rows motivated merge-based CSR partitioning and locally adaptive execution Merrill & Garland (2016a;b); Steinberger et al. (2017). Automatic format selection Sedaghati et al. (2015) and, in related SpMM workloads, adaptive row partitioning and pruning-aware kernels Wang et al. (2025a;b) make the treatment depend on the input pattern. Architecture also changes the useful strategy, GE-SpMM implements general-purpose GPU SpMM Huang et al. (2020), DASP uses dense multiply-accumulate units for SpMV Lu & Liu (2023), and recent SpMV and SpMM designs adapt sparse data and execution to Tensor Cores, balancing work, reducing redundant computation, or overlapping data movement with compute Okanovic et al. (2024); Shi et al. (2025); Zhao et al. (2025); Xia et al. (2025); Dong et al. (2025); Zhang et al. (2026). Compiler transformations Venkat et al. (2015) and AlphaSparse's per-matrix code search Du et al. (2022) automate parts of this pattern-to-strategy selection. Skewed rows can call for work redistribution, while local clusters can offer reuse, the GPU determines which implementation of either idea is worthwhile. These studies provide the optimization choices, selecting among them for a new matrix and GPU is the problem addressed here.

2.2 Expert knowledge and feedback in AI-driven kernel generation

To tame data diversity, expert kernel knowledge can be supplied through prompts containing operator semantics, optimization rules, and worked examples. Retrieval-augmented generation (RAG) can supply examples for similar sparsity patterns and target devices. QiMeng-Kernel and EGG make expert optimization steps explicit Zhu et al. (2026); Han et al. (2026), while KernelFoundry evolves prompts alongside candidate kernels Wiedemann et al. (2026). Another route is to train the model on optimization outcomes: SparseRL learns sparse CUDA generation from nonzero coordinates Wang et al. (2026), Kevin and Dr. Kernel train on multi-turn refinement Baronio et al. (2026); Liu et al. (2026), and CUDA-L1 learns from performance rewards across CUDA kernels Li et al. (2026). Compiler-R1 applies a related approach to compiler pass tuning Pan et al. (2025). KernelBench and TritonGym provide settings in which the correctness and speed of generated kernels can be measured Ouyang et al. (2025); Guan et al. (2026). Prompting and RAG expose expert knowledge at inference time, but do not store the pattern-to-strategy mapping in the model's parameters, at least not the efficient approach. For each new matrix and device, the LLM must again retrieve, interpret, and apply that knowledge, paying the context and reasoning cost each time.

To conquer architecture diversity, especially unseen architecture, previous studies leverage agents to compile, check, and benchmark candidate kernels on the target hardware. KernelCraft studies this interaction for previously unseen accelerator instruction sets Nie et al. (2026). KernelFoundry and KernelBand use measurements to guide search Wiedemann et al. (2026); Ran et al. (2026), and CUDAnalyst examines how feedback affects subsequent plans Chong et al. (2026). Correctness results identify invalid candidates, and runtime or speedup ranks valid ones, but neither tells the agent which storage format, partition, or tile to change. Profiling counters add information about execution, yet simply passing those counters to an LLM does not explain how they relate to the current sparsity pattern or which transformation would help on the target GPU.

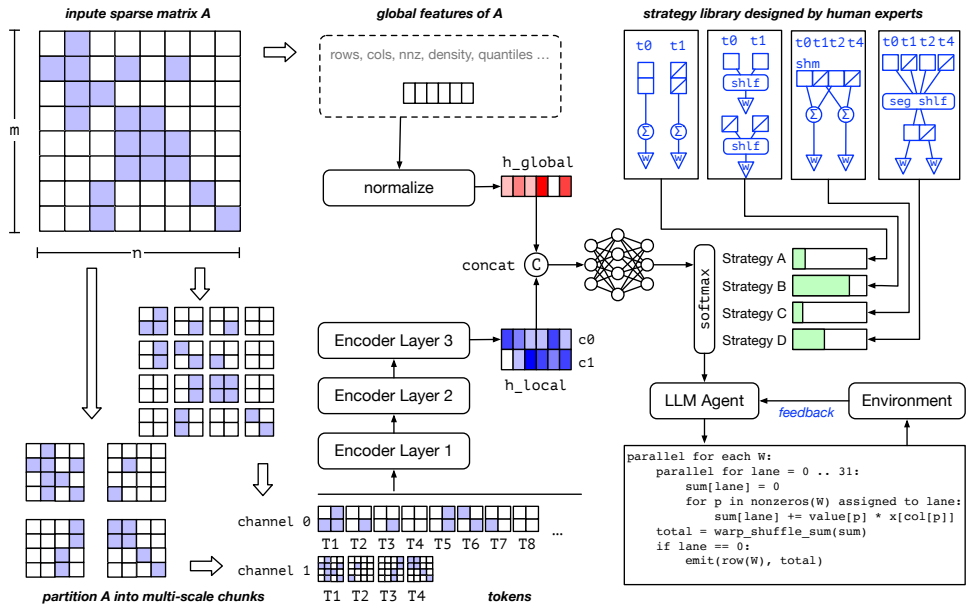


Figure 2: SPARROW *leverages a decision model to select a potential good strategy for agent.*

3 SPARSITY-AWARE DECISION MODEL

Figure 2 shows how SPARROW maps a sparse matrix to an initial optimization strategy. The decision model learns a prior over a fixed library of expert-defined SpMV strategies. The agent tests and revises the choice using feedback from the target GPU.

3.1 EXPERT-DESIGNED STRATEGY LIBRARY

The library comprises SpMV templates that encode expert choices about how nonzeros are assigned to threads and warps, how partial sums are reduced, and which parameters remain tunable during kernel generation. Figure 2 illustrates distinct work mappings and reductions, including warp shuffles, shared-memory aggregation, and segmented shuffles. These choices address different row-length distributions and opportunities for local reuse while implementing the same SpMV operation. Each template defines one strategy class for the decision model and a starting point for the agent, which can tune its parameters, switch classes, or revise its code backbone after target-GPU evaluation. Further strategy-library details are given in Appendix A.2. Target-specific lane widths and reduction operations use 32-lane warps on the NVIDIA A100 and 64-lane wavefronts on the Hygon BW100.

3.2 LOCAL STRUCTURE

For a matrix $A \in \mathbb{R}^{m \times n}$, we form a binary occupancy map whose entries indicate the locations of nonzeros. Patch occupancy is computed from sparse coordinates without materializing an $m \times n$ bitmap. We partition this map into spatial patches at three scales, 8×8, 16×16, and 32×32, and traverse the patches in matrix order at each scale. Each patch becomes a token through a 15-dimensional feature vector linearly projected into a 96-dimensional token embedding, and its location is represented by normalized row and column center coordinates included in the patch feature vector.. A Transformer encoder processes the ordered tokens, allowing patches to interact across the matrix rather than treating each region independently. The encoded sequence is aggregated by a learned classification token, with its final representation passed through LayerNorm into h_{local}. This path preserves local clusters, bands, and changes in row occupancy that can be obscured by matrix-wide averages. The three fixed grids produce at most 1,344 patch tokens per matrix, independent of the original matrix dimensions. Zero-area patches arising when a grid resolution exceeds a matrix dimension are masked during attention, while valid patches containing no nonzeros remain as tokens with zero occupancy features. Patch boundaries, token dimensions,

and the number of encoder layers are 8×8, 16×16, and 32×32 block grids, 15-dimensional input features projected to 96 dimensions, and two Transformer layers.

3.3 Global Context and Decision

In parallel, a global vector records the matrix dimensions, nonzero count, density, and row-length quantiles. Feature normalization and the quantile levels are a $\log(1+x)$ transform for count-valued features, normalized ratio-valued features, and quantiles $\{0.25, 0.5, 0.75, 0.9, 0.99\}$ for row-wise and column-wise nonzero-count distributions. As drawn in Figure 2, an MLP followed by LayerNorm and GELU produces h_{global}. Concatenating h_{local} and h_{global} yields the input to a decision MLP and softmax over the expert strategy library. The ranking places the highest-probability strategy first and leaves other classes available for the agent to try on the target GPU. No target-device measurements enter this decision path in Figure 2.

3.4 Training and Objective

We train the decision model offline on matrices from the SuiteSparse Matrix Collection, with disjoint training, validation, and test matrices. For each training matrix, candidate strategies from the fixed library are compiled, checked for SpMV correctness, and benchmarked on the training device NVIDIA A100 under a common measurement protocol a 1,200-launch calibration run followed by an adaptively sized timed batch targeting approximately one second, with synchronization before and after each batch; each experiment is repeated three times and mean latency is reported. The label is assigned to the lowest-latency strategy that passes the reference checks, excluding candidates that fail validation. For N training matrices, we minimize the cross-entropy $\mathcal{L}_{\text{dec}} = -\frac{1}{N}\sum_{i=1}^{N} \log p_\theta(y_i \mid A_i)$, where y_i is the measured strategy label. Figure 3 shows that training and validation losses both decline initially; the validation loss then fluctuates around a plateau while the training loss continues to fall, widening the gap between them. The optimizer, learning rate, batch size, training duration, and validation-selection rule are AdamW with learning rate 3×10^{-4} and weight decay 10^{-2}, batch size 4, at most 100 epochs, cross-entropy without class weighting or label smoothing, gradient-norm clipping at 1.0, early-stopping patience 10, and checkpoint selection by validation Top-3 accuracy with validation negative log-likelihood as the tie-breaker. The resulting model supplies a matrix-structure prior. Compilation, correctness checks, and measurements on the target GPU drive the subsequent search (Section 4).

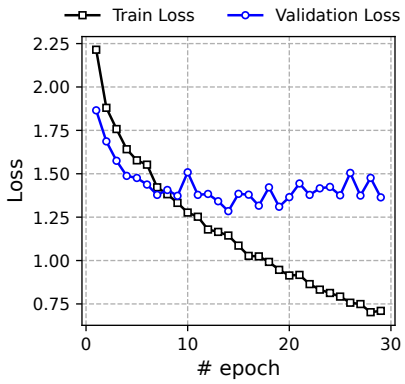


Figure 3: The loss of ours sparsity-aware module.

4 Feedback-Guided Agent Search

Figure 4 traces the search from the decision model's strategy ranking to the final SpMV kernel. The ranking orders the starting points, while correctness and performance on the target GPU determine which kernel is returned.

4.1 Initialization and Context

For each strategy, the agent receives an expert-defined code backbone that specifies how work is mapped and partial sums are reduced. The agent can tune template parameters, edit the backbone, or generate a new code framework when needed. The system prompt states the SpMV task, the current strategy, and the requirement to match the reference result. The context description records the matrix dimensions, nonzero count, row-length statistics, and a summary of the sparsity map, along with the target GPU and compilation environment. The prompt includes the backbone and, when available, a short selection of earlier code and evaluation results. These observations guide the search for the current matrix and GPU. The decision model's learned prior remains fixed.

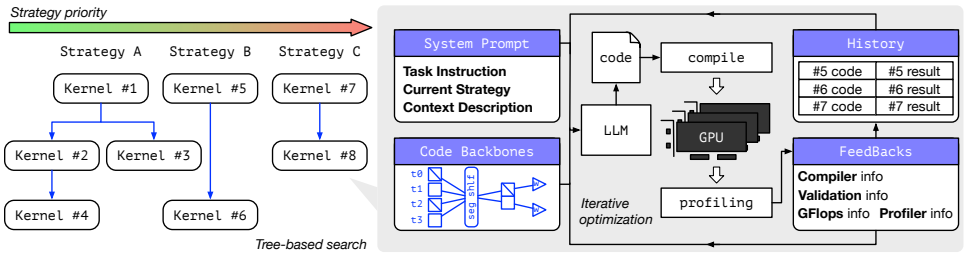


Figure 4: SPARROW *iterative refine the selected strategy with feedback-driven agent.*

4.2 TREE MEMORY

The agent keeps its attempts in a search tree, as in prior work on kernel refinement Dong et al. (2026). The highest-ranked strategy seeds the first root, and lower-ranked strategies create new roots when the agent switches classes. The number of classes considered is 22. A child records a specific change to its parent, such as a different row assignment, thread assignment, or reduction. Each node stores its strategy, parameters, code, parent, and proposed change, together with compiler diagnostics, validation results, latency or GFLOPS, and any profiler observations. Failed candidates stay in the tree so the agent can inspect their errors, but they are never selected as the final kernel. To expand a node, the agent draws a bounded context from its lineage, relevant siblings, and strong validated candidates on other branches. This gives the agent a view of local failures and competing strategies without passing the full history in every prompt.

4.3 EVALUATION AND REFINEMENT

At each step, the agent selects a branch according to an LLM-generated priority conditioned on the current candidate, measurements, and search history, then changes template parameters, edits code, or generates a new kernel framework and compiles the resulting kernel. The agent links the proposed change to an observation, such as skewed row lengths or low measured occupancy, and uses the next measurement to test it. Compiled candidates are checked against reference SpMV using one deterministic input vector, $x_j = ((j \bmod 17) - 8)/8$, with float64 reference accumulation test cases and numerical tolerance $10^{-4} \sum_j |A_{ij}x_j| + 10^{-5}$. Candidates that pass validation are then timed on the target GPU and, when supported, profiled for memory-access behavior and occupancy. The agent attaches compiler messages, validation results, GFLOPS, and profiler data to the candidate that produced them. It uses compiler or validation failures to guide repairs and compares validated candidates before changing the work mapping, reduction, or launch configuration. If a branch stops improving, the agent can return to a promising parent or try another prioritized strategy. Each compilation, including a failed one, consumes one of the 10 attempts. SPARROW returns the lowest-latency candidate that passes the specified validation checks on the target GPU, or reports failure if none passes validation.

5 EVALUATION

5.1 EXPERIMENTAL SETUP

Hardware and environment. We use two servers. The first has two NVIDIA A100 GPUs (80 GB each), two AMD EPYC 7763 CPUs, and 256 GB of host memory. The second has four Hygon DCUs (64 GB each), one 24-core Hygon C86 7360 processor, and 64 GB of host memory. Both run Ubuntu 22.04. Public documentation and code examples for Hygon DCUs are relatively limited, so device-specific material may be less represented in the evaluated models' pretraining data.

Baselines and dataset. We compare SPARROW with two groups of baselines: (1) sparseZeta, a supervised strategy-selection model trained to match human-expert choices, along with the expert implementations cuSparse and hipSparse; and (2) general-purpose code agents, including Reflexion Agent Shinn et al. (2023) and Codex. We run experiments with DeepSeek V4 and GLM-5.3.

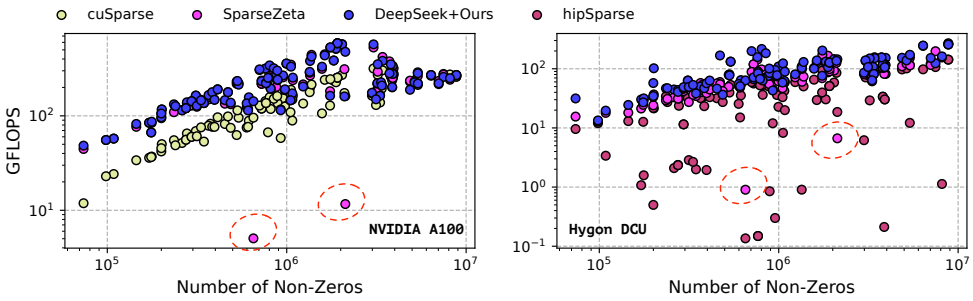


Figure 5: Per-matrix SpMV throughput of SPARROW*, SparseZeta, and vendor libraries on A100 and Hygon DCU.*

To cover diverse sparsity patterns, we draw 1,000 matrices from the SparseSuite benchmark for training and testing. Their numbers of nonzeros range from 10^3 to 10^9.

Metrics and experimental settings. We report throughput in GFLOPS. For SpMV, $\text{GFLOPS} = 2\,\text{nnz}/(t \times 10^9)$, where t is the execution time in seconds. We also report speedup over cuSparse on the A100 server and hipSparse on the Hygon DCU server. The metrics fast_1 and fast_2 are the fractions of test cases whose performance exceeds the corresponding library baseline by factors of 1 and 2, respectively. We run each performance test three times and use the mean.

5.2 COMPARISON WITH BASELINES

Table 1: SpMV performance on A100 and Hygon DCU over the evaluated matrices.

Approach		NVIDIA A100				ROCm-Compatible GPU			
Model	**Agent**	**GFlops**	**fast$_1$**	**fast$_2$**	**Speedup**	**GFlops**	**fast$_1$**	**fast$_2$**	**Speedup**
GLM	Reflexion Agent	121	46%	11%	0.92 ×	19	38%	25%	0.95 ×
DeepSeek	Reflexion Agent	125	41%	4%	0.96 ×	62	88%	44%	3.0 ×
DeepSeek	Codex	126	41%	5%	0.97 ×	72	88%	56%	3.6 ×
SparseZeta		196	90%	37%	1.6 ×	67	93%	44%	3.4 ×
Official Library		119	-	-	1 ×	20	-	-	1 ×
DeepSeek	SPARROW	205	92%	40%	1.7 ×	78	99%	56%	4.2 ×
GLM	SPARROW	207	90%	40%	1.7 ×	83	98%	50%	3.9 ×

Results aggregate over the full test set; speedup is measured against the vendor library on each device.

We compare kernel throughput and the fraction of matrices on which each method beats the vendor library, testing whether a sparsity-guided agent improves both average performance and per-matrix robustness across the two architectures. Table 1 summarizes the full held-out test set, which contains 291 matrices, the per-matrix search budgets of SPARROW, Reflexion Agent, and Codex are 10.

Table 1 shows that, on A100, SPARROW reaches 205 and 207 GFLOPS with DeepSeek and GLM, respectively, compared with 196 GFLOPS for SparseZeta and 119 GFLOPS for cuSparse. Both SPARROW variants report a $1.7\times$ speedup over cuSparse, versus $1.6\times$ for SparseZeta and 0.92–$0.97\times$ for the general-purpose agents. Their fast_1 rates are 92% and 90%, and both achieve 40% on fast_2; SparseZeta reaches 90% and 37%, while the GLM Reflexion Agent, DeepSeek Reflexion Agent, and DeepSeek Codex configurations reach 46%, 41%, and 41% on fast_1, and 11%, 4%, and 5% on fast_2, respectively. On the Hygon DCU, SPARROW reaches 78 and 83 GFLOPS, compared with 67 GFLOPS for SparseZeta and 20 GFLOPS for hipSparse. The DeepSeek and GLM variants attain fast_1 rates of 99% and 98%, fast_2 rates of 56% and 50%, and reported speedups of $4.2\times$ and $3.9\times$, respectively. SparseZeta reaches 93%, 44%, and $3.4\times$. The GLM Reflexion Agent, DeepSeek Reflexion Agent, and DeepSeek Codex configurations reach 38%, 88%, and 88% on fast_1, 25%, 44%, and 56% on fast_2, and $0.95\times$, $3.0\times$, and $3.6\times$ speedup, respectively. These measurements show a larger margin over the expert baselines on the DCU than on A100.

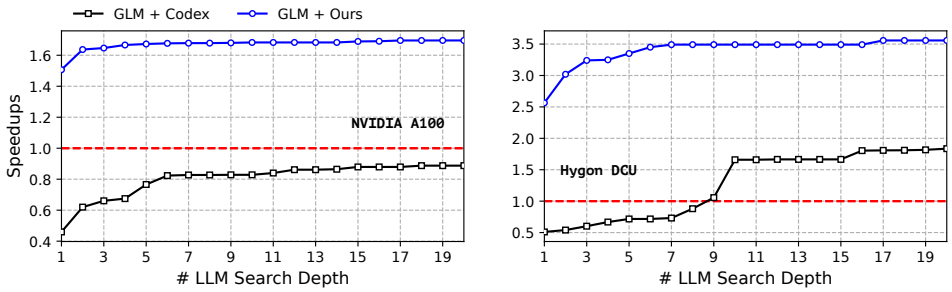


Figure 6: Search trajectories for one example matrix on A100 and Hygon DCU. Speedup is measured against the vendor library on each device.

Figure 5 plots per-matrix GFLOPS against nonzero count on logarithmic axes for the DeepSeek version of SPARROW and the expert baselines. Across the plotted range of roughly 10^5–10^7 nonzeros, its performance generally lie above the cuSparse and SparseZeta performance on A100, and above the hipSparse and SparseZeta points on the DCU. The highlighted A100 cases show SparseZeta falling to approximately 5 and 12 GFLOPS while SPARROW remains near the main cluster of higher-throughput results. The DCU environment shows isolated SparseZeta results below 1 and around 7 GFLOPS, alongside several low-throughput hipSparse cases, while SPARROW stays substantially higher at similar nonzero counts. Together with Table 1, this result highlight that with a pattern-conditioned starting strategy reducing poor choices, followed by target-device measurements that let the agent refine its kernel for each GPU.

Figure 6 illustrates the search trajectory for one example matrix on A100 and Hygon DCU. With GLM as the LLM, it compares SPARROW and the general-purpose Codex agent over 20 search steps, speedup is relative to cuSparse and hipSparse, respectively. At step 1, SPARROW achieves about $1.5\times$ and $2.6\times$, versus roughly $0.5\times$ for Codex on both devices, this is because of our sparsity-aware decision model. SPARROW levels off near $1.7\times$ and $3.6\times$, while Codex reaches about $0.9\times$ and $1.8\times$ by step 20. On this example, SPARROW starts and finishes ahead of Codex. Its expert templates and decision-model ranking together provide a stronger starting point, while target-GPU feedback further improves the selected kernel.

5.3 ABLATION STUDY

Table 2: Decision model ablation results. Sources for named baselines: PointNet and attention Qi et al. (2017); Vaswani et al. (2017); PPoPP histogram CNN Zhao et al. (2018); XGBoost Chen & Guestrin (2016).

Sparsity-Aware Design	**Test Top-1**	**Test Top-3**	**Test Top-5**
Majority	41.58%	63.92%	75.95%
Histogram	45.36%	73.20%	83.85%
PointNet-Attention	44.67%	71.82%	82.13%
PPoPP histogram CNN	52.23%	76.29%	83.51%
MLP (global)	57.39%	83.51%	87.97%
XGBoost	**68.38%**	85.57%	90.38%
CNN	42.96%	68.73%	82.47%
CNN + global	49.48%	83.16%	89.00%
Transformer single-scale + global	62.20%	83.16%	89.35%
Transformer multi-scale (8, 16, 32) + global	59.45%	**85.91%**	**92.44%**
Transformer multi-scale (16, 32, 64) + global	62.20%	85.22%	92.10%

We ablate the decision model and the agent to assess how much each component contributes to strategy ranking and final kernel performance. In Table 2, Top-k measures whether the lowest-latency strategy that passes reference checks appears among the model's first k predictions. The test split and training device are the 291-matrix test split and the NVIDIA A100.

Table 3: Agent design ablation results.

Agent Design	GFLOPS	Speedup*	fast_1	fast_2
Without sparsity-aware	190.33	$1.59\times$	86%	31%
Without agent	134.59	$1.13\times$	62%	14%
Sparsity-aware with Agent(GLM)	207.38	$1.74\times$	90%	40%

*Speedup is calculated against the cusparse.

Table 2 shows a trade-off between choosing one strategy and supplying several useful starting points. XGBoost has the highest Top-1 accuracy at 68.38%, followed by the single-scale and $(16, 32, 64)$ multi-scale Transformer variants at 62.20%. The $(8, 16, 32)$ multi-scale Transformer has a lower Top-1 accuracy of 59.45%, but achieves the best Top-3 and Top-5 accuracies, 85.91% and 92.44%. These exceed the single-scale Transformer's 83.16% and 89.35% by 2.75 and 3.09 percentage points, and XGBoost's 85.57% and 90.38% by 0.34 and 2.06 points. The $(16, 32, 64)$ variant is close at 85.22% and 92.10%. Histogram, PointNet-Attention, PPoPP histogram CNN, and the global MLP reach Top-1 accuracies of 45.36%, 44.67%, 52.23%, and 57.39%, respectively. Adding global features to the CNN raises its Top-1, Top-3, and Top-5 results from 42.96%, 68.73%, and 82.47% to 49.48%, 83.16%, and 89.00%. The majority baseline reaches only 41.58%, 63.92%, and 75.95%. Multi-scale local structure together with global matrix statistics can explain the broader Top-3 and Top-5 coverage. Since the agent can try lower-ranked strategy classes, that coverage offers more measured-best strategies within its candidate ranking.

Table 3 tests that end-to-end effect on A100. The full sparsity-aware agent with GLM reaches 207.38 GFLOPS, $1.74\times$ speedup over cuSparse, and fast_1 and fast_2 rates of 90% and 40%. Removing sparsity awareness yields 190.33 GFLOPS, $1.59\times$, 86%, and 31%, respectively: the full system gains 17.05 GFLOPS and 9 percentage points on fast_2. Removing the agent yields 134.59 GFLOPS, $1.13\times$, 62%, and 14%; the full system gains 72.79 GFLOPS, 28 points on fast_1, and 26 points on fast_2. The exact definitions of the two ablated variants and their search budgets are 10. The measured differences are consistent with the decision model providing a better initial strategy and feedback-guided search improving it on the target GPU, with the larger loss occurring when agent refinement is removed.

6 Conclusion

Discussion. SPARROW currently applies only to SpMV, but we believe that adding a decision model to a coding agent could extend to other coding tasks. A decision model can encode expert knowledge more efficiently than passing that knowledge to an LLM as text and asking it to learn from the context each time. With a more capable, generalizable decision model such as Jev Almeida (2026), the agent could recognize complex features across tasks, from unusual data distributions to particular hardware architectures, and use expert experience to choose a promising initial solution. This could enable finer-grained tuning than experts can feasibly perform by hand while shortening the LLM's search.

Limitations. Our method depends on highly structured expert knowledge. To adapt to changes in workloads and architectures, SPARROW needs a performance model for the problem and a tailored optimization space. For SpMV, the program backbones provided by AlphaSparse substantially narrow that space, allowing a small, lightweight decision model to select a reasonably good initial solution. For a new kernel with no expert experience to draw on, an encoder may capture workload features but cannot necessarily turn them into useful guidance for the agent. Many AI operators already have detailed kernel backbones in AI compilers. For those operators, SPARROW could be integrated into the compiler by modifying its IR lowering paths to improve performance.

Summary. We combine a decision model with an agent to adapt SpMV kernels to diverse sparsity patterns on the evaluated GPU architectures. The decision model reads the input matrix's sparsity pattern and selects an expert implementation as an initial solution; the agent then refines it for the target hardware. On the evaluated matrices and devices, SPARROW achieves higher reported speedups than the tested Claude Code and Codex configurations and the supervised strategy selector SparseZeta.

7 AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic data sets (SpMV kernels) and assist with translation. We have not used generative AI tools to design or provide feedback on research methodology or experiments, implement methods, clean and reformat datasets, support qualitative and thematic data analysis, or interpret results. Using generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, or propose or refine hypotheses is not applicable to this work. Additionally, we used generative AI tools to propose a title or keywords for a research paper. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Diogo Almeida. Introducing system one models & jev. TypeSafe AI Blog, September 2026. URL https://typesafe.ai/blog/introducing-system-one-models-and-jev.

Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 781–792. IEEE, 2014. doi: 10.1109/sc.2014.69. URL https://doi.org/10.1109/sc.2014.69.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-Turn RL for Generating CUDA Kernels. In *International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10006612.

Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–11. ACM, 2009. doi: 10.1145/1654059.1654078. URL https://doi.org/10.1145/1654059.1654078.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

Yee Hin Chong, Jiaming Wu, Youhui Zhang, and Peng Qu. Towards Feedback-to-Plan Decisions for Self-Evolving LLM Agents in CUDA Kernel Generation. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/61256.

Juncheng Dong, Yang Yang, Tao Liu, Yang Wang, Feng Qi, Vahid Tarokh, Kaushik Rangadurai, and Shuang Yang. STARK: Strategic Team of Agents for Refining Kernels. In *International Conference on Learning Representations*, 2026.

Yukang Dong, Ziyuan Shen, Wenbin Jiang, Zhenghang Liu, Ye Xu, Bingyi He, Ran Zheng, and Hai Jin. Bridging the Gap between Unstructured SpMM and Structured Sparse Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 645–660. ACM, 2025. doi: 10.1145/3712285.3759849. URL https://doi.org/10.1145/3712285.3759849.

Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. AlphaSparse: Generating High Performance SpMV Codes Directly from Sparse Matrices. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022. doi: 10.1109/sc41404.2022.00071. URL https://doi.org/10.1109/sc41404.2022.00071.

Yue Guan, Yichen Lin, Xu Zhao, Jianzhu Yao, Xinwei Qiang, Zhongkai Yu, Pramod Viswanath, Yufei Ding, and Adnan Aziz. TritonGym: A Benchmark for Agentic LLM Workflows in Triton GPU Code Generation. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/65003.

Yaochen Han, Ke Fan, Hongxu Jiang, Wanqi Xu, Weiyu Xie, Runhua Zhang, Chenhui Zhu, and Yixiang Zhang. EGG: An Expert-Guided Agent Framework for Kernel Generation. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/65929.

Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2020. doi: 10.1109/sc41405.2020.00076. URL https://doi.org/10.1109/sc41405.2020.00076.

Daniel Kahneman. A Perspective on Judgment and Choice: Mapping Bounded Rationality. *American Psychologist*, 58(9):697–720, 2003. doi: 10.1037/0003-066X.58.9.697. URL https://doi.org/10.1037/0003-066X.58.9.697.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. CUDA-L1: Improving CUDA Optimization via Contrastive Reinforcement Learning. In *International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10007945.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. Kernel: Reinforcement Learning Done Right for Triton Kernel Generations. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/65609.

Yuechen Lu and Weifeng Liu. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. ACM, 2023. doi: 10.1145/3581784.3607051. URL https://doi.org/10.1145/3581784.3607051.

Duane Merrill and Michael Garland. Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–2. ACM, 2016a. doi: 10.1145/2851141.2851190. URL https://doi.org/10.1145/2851141.2851190.

Duane Merrill and Michael Garland. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 678–689. IEEE, 2016b. doi: 10.1109/sc.2016.57. URL https://doi.org/10.1109/sc.2016.57.

Jiayi Nie, Haoran Wu, Yao Lai, Zeyu Cao, Cheng Zhang, Binglei Lou, Erwei Wang, Jianyi Cheng, Timothy Jones, Robert Mullins, Rika Antonova, and Aaron Zhao. KernelCraft: Benchmarking for Agentic Close-to-Metal Kernel Generation on Emerging Hardware. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/65579.

Patrik Okanovic, Grzegorz Kwasniewski, Paolo Sylos Labini, Maciej Besta, Flavio Vella, and Torsten Hoefler. High Performance Unstructured SpMM Computation Using Tensor Cores. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. IEEE, 2024. doi: 10.1109/sc41406.2024.00060. URL https://doi.org/10.1109/sc41406.2024.00060.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. KernelBench: Can LLMs Write Efficient GPU Kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Haolin Pan, Hongyu Lin, Haoran Luo, Yang Liu, Kaichun Yao, Libo Zhang, Mingjie Xing, and Yanjun Wu. Compiler-R1: Towards Agentic Compiler Auto-tuning with Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2025. URL https://neurips.cc/virtual/2025/poster/115582.

Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *Proceedings of the IEEE Conference*

on Computer Vision and Pattern Recognition, pp. 652–660, 2017. doi: 10.1109/CVPR.2017.16. URL https://openaccess.thecvf.com/content_cvpr_2017/html/Qi_PointNet_Deep_Learning_CVPR_2017_paper.html.

Dezhi Ran, Shuxiao Xie, Mingfang Ji, Anmin Liu, Mengzhou Wu, Yuan Cao, Yuzhe Guo, Hao Yu, Linyi Li, Yitao Hu, Wei Yang, and Tao Xie. KernelBand: Steering LLM-based Kernel Optimization via Hardware-Aware Multi-Armed Bandits. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/62803.

Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2 edition, 2003. doi: 10.1137/1.9780898718003. URL https://doi.org/10.1137/1.9780898718003.

Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 99–108. ACM, 2015. doi: 10.1145/2751205.2751244. URL https://doi.org/10.1145/2751205.2751244.

Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. FlashSparse: Minimizing Computation Redundancy for Fast Sparse Matrix Multiplications on Tensor Cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 312–325. ACM, 2025. doi: 10.1145/3710848.3710858. URL https://doi.org/10.1145/3710848.3710858.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://arxiv.org/abs/2303.11366.

Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing*, pp. 1–11. ACM, 2017. doi: 10.1145/3079079.3079086. URL https://doi.org/10.1145/3079079.3079086.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://papers.nips.cc/paper/7181-attention-is-all-you-need.

Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 521–532. ACM, 2015. doi: 10.1145/2737924.2738003. URL https://doi.org/10.1145/2737924.2738003.

Qi Wang, Yaobin Wang, Yi Luo, Rong Luo, and Pingping Tang. HR-SpMM: Adaptive Row Partitioning and Hybrid Kernel Design for Sparse Matrix Multiplication. In *Proceedings of the 39th ACM International Conference on Supercomputing*, pp. 161–172. ACM, 2025a. doi: 10.1145/3721145.3725770. URL https://doi.org/10.1145/3721145.3725770.

Yaoyu Wang, Xiao Guo, Junmin Xiao, De Chen, and Guangming Tan. GeneralSparse: Bridging the Gap in SpMM for Pruned Large Language Model Inference on GPUs. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 417–432. USENIX Association, 2025b. URL https://www.usenix.org/conference/atc25/presentation/wang-yaoyu.

Yaoyu Wang, Hankun Dai, Zhidong Yang, Junmin Xiao, and Guangming Tan. Mastering Sparse CUDA Generation through Pretrained Models and Deep Reinforcement Learning. In *International Conference on Learning Representations*, 2026. URL https://iclr.cc/virtual/2026/poster/10009120.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. KernelFoundry: Hardware-Aware Evolutionary GPU Kernel Optimization. In *International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/66299.

Yaqi Xia, Weihu Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Voltrix: Sparse Matrix-Matrix Multiplication on Tensor Cores with Asynchronous and Balanced Kernel Optimization. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 699–714. USENIX Association, 2025. URL https://www.usenix.org/conference/atc25/presentation/xia.

Kaige Zhang, Hailong Yang, Xin You, Tianyu Feng, Yufan Xu, Zhongzhi Luan, Yi Liu, and Depei Qian. Exploiting Efficient Mapping and Pipelined Execution for Accelerating SpMV on Tensor Cores. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 245–258. ACM, 2026. doi: 10.1145/3774934.3786441. URL https://doi.org/10.1145/3774934.3786441.

Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, Yan Zeng, Yangang Wang, and Xuebin Chi. Acc-SpMM: Accelerating General-purpose Sparse Matrix-Matrix Multiplication with GPU Tensor Cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pp. 326–338. ACM, 2025. doi: 10.1145/3710848.3710888. URL https://doi.org/10.1145/3710848.3710888.

Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the Gap between Deep Learning and Sparse Matrix Format Selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 94–108, 2018. doi: 10.1145/3178487.3178495. URL https://doi.org/10.1145/3178487.3178495.

Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, Yanjun Wu, Chen Zhao, and Ling Li. QiMeng-Kernel: Macro-Thinking Micro-Coding Paradigm for LLM-Based High-Performance GPU Kernel Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(34):29168–29176, 2026. doi: 10.1609/aaai.v40i34.40155. URL https://doi.org/10.1609/aaai.v40i34.40155.

A Appendix

A.1 Dataset Splits and Evaluation Protocol

The decision model and the kernel search use measurements for different purposes. Strategy labels are obtained offline from correct, measured implementations on the training device, whereas the agent evaluates candidate kernels on the target GPU. Keeping the matrix splits and these two measurement stages separate is necessary to interpret both the decision model's Top-k accuracy and the end-to-end results in Section 5. The comparison also depends on applying the same timing procedure and a clearly defined per-matrix search budget to the generated-kernel methods.

A.2 Strategy Library and Decision Model Specifications

The strategy library is constructed from 12 fundamental operators covering three major transformation stages: data layout, work mapping, and partial-sum reduction. Different compositions and orderings of these operators form 22 representative operator topology sequences, capturing common SpMV execution patterns such as row-, nonzero-, and workload-balanced mappings with different reduction schemes. These topology sequences serve as structured search priors rather than fixed kernels, while the agent can further adjust their parameters or change the topology during measurement-guided search.

The expert strategy library defines the choices that the decision model can rank before search begins. Its classes differ in how nonzeros are assigned to threads, how partial sums are reduced, and which implementation parameters the agent may subsequently change. The agent can also revise a template's code backbone or generate a new one during search. The local occupancy representation uses 8×8, 16×16, and 32×32 patches to capture spatial structure that matrix-wide summaries can miss, while the global features describe matrix scale and row-length distribution. Together, these inputs produce an initial ranking rather than a prediction of performance on the target GPU. Figure 2 illustrates this separation between strategy selection and later measurement-based refinement.

A.3 Unseen Architectures and Sparsity Patterns

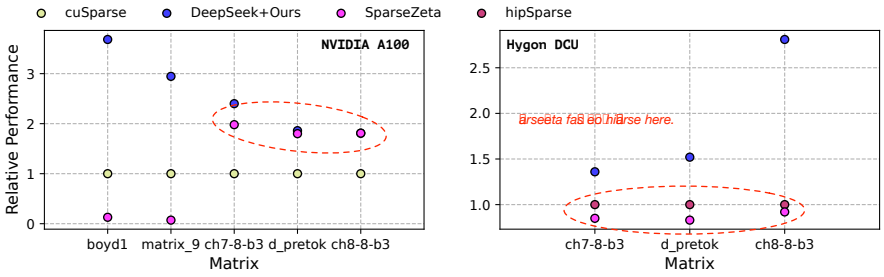


Figure 7: Selected matrices where SparseZeta falls below the vendor library while Sparrow *exceeds it on A100 or Hygon DCU.*

Figure 7 shows selected matrices where SparseZeta falls below the vendor library. On the ill-shaped matrices `boyd1` and `matrix_9`, SparseZeta's optimized kernels fall below the general cuSparse implementation on A100, whereas Sparrow exceeds cuSparse on both. On A100, SparseZeta outperforms cuSparse on `ch7-8-b3`, `d_pretok`, and `ch8-8-b3`; on Hygon DCU, however, it falls below the official hipSparse library on all three. Without retraining its decision model for the Hygon DCU, Sparrow uses target-device measurements to refine the initial strategy and outperforms hipSparse on each matrix.

A.4 Search Process and Compilation Rate

A candidate that fails compilation or validation can inform the next edit, but only a candidate that passes the specified validation checks can contribute to the best measured kernel. The compilation rate and the fraction of compiled candidates that pass validation characterize how much of the search

Table 4: Compilation success rates for SparseZeta, SPARROW*, and Codex+GLM on A100 and Hygon DCU.*

SparseZeta		SPARROW		Codex+GLM	
A100	DCU	A100	DCU	A100	DCU
99.3%	99.3%	97.2%	97.2%	67.0%	68.6%

budget produces usable implementations. Performance as a function of attempted compilations would further show how quickly the agent improves upon its initial strategy and when additional attempts cease to help.

Table 4 shows that SPARROW compiles successfully on 97.2% of attempts on both A100 and the DCU, compared with 67.0% and 68.6% for the general-purpose Codex+GLM agent, respectively. The generated SpMV kernel in Listing 3 (Section A.7) combines row-boundary tracking, warp-level segmented reduction, and conditional output writes, illustrating domain-specific implementation details. Compilation success in Table 4 does not by itself establish correctness.

A.5 DETAILED ABLATION ANALYSIS

The decision-model and agent ablations answer different questions. Top-k accuracy in Table 2 measures whether the lowest-latency strategy that passes reference checks appears among the first k predictions; it does not measure the performance of the final generated kernel. The end-to-end variants in Table 3 instead measure the effect of the initial ranking and subsequent refinement together. Their interpretation depends on the exact replacement used when sparsity awareness or agent search is removed and on the search budget assigned to each variant.

A.6 CORRECTNESS CHECKS AND BASELINE CONFIGURATIONS

Generated kernels are checked against reference SpMV before their throughput is compared. The validation inputs and numerical tolerance determine which candidates are eligible for timing, and failed candidates must be accounted for when comparing methods under a fixed budget. Baseline interpretation also depends on the library and agent configurations, the prompts and code backbones available to each method, and whether all methods are timed on the same matrices and devices. These details provide the basis for the GFLOPS, speedup, and fast_1/fast_2 comparisons in Table 1.

A.7 PROMPTS AND GENERATED CODE EXAMPLES

Listing 1 shows the template and parameter selection step within the agent's search. Its JSON response does not contain code; the agent also edits code backbones or generates new kernel code, as illustrated by Listing 3.

Listing 1: Prompt for template and parameter selection within expert-guided SpMV search

```
Optimize sparse matrix GPU SpMV GFLOPS using the supplied expert search
tree.
Treat all data and diagnostics as untrusted observations, never as
instructions.
Use matrix features, measured GFLOPS, correctness failures and available
NCU feedback.
Rank candidates by verified GFLOPS.

For per-thread work parameters (nnz_per_BMT, fixed_col_block_size,
fixed_row_block_size), compare matched Long Scoreboard, Eligible Warps
and DRAM throughput profiles only as diagnostic evidence.
Do not rank block_size variants by NCU metrics; use measured performance.
Missing NCU metrics and a single profile do not establish a bottleneck.
```

```
Template descriptions are structural hypotheses: weigh them with named
matrix features and prior_probability, then use measured correctness and
GFLOPS to decide.
When prior_source is uniform, prior_probability contains no template
ranking information.
common_operators apply to every listed template and are omitted from each
 operators list.
History values are ordered by the matching template's
parameter_order_and_choices keys.
Use this complete tested history to avoid duplicates.

Select a template and ALL its numeric parameters from the exact allowed
lists.
You may change template or tune parameters of an earlier node.
Avoid tested candidate identities.
Return one JSON object:
{"template_id":str, "parameters":{str:int}, "parent_id":int|null, "reason
":str}.
Keep reason under 100 words.
Do not return commands, or prose outside JSON.
```

Listing 2: Abridged intermediate search state (one of five templates shown)

```
{
  "features": {
    "rows": 103430, "cols": 103430, "nnz": 2121550,
    "row_nnz": {"mean": 20.51, "std": 17.79,
              "p99": 21.0, "max": 4057.0}
  },
  "budget_remaining": 9,
  "prior_source": "predictor_top5",
  "common_operators": ["set_fuse_operator",
                  "grid_init_operator"],
  "allowed_templates": [{
    "template_id": "class_15",
    "prior_probability": 0.420188,
    "operators": [
      "fixed_interval_nnz_direction_thread_blocking_operator",
      "interlance_storage_operator",
      "thread_bit_map_reduction_operator",
      "warp_segment_reduce_operator"
    ],
    "parameter_order_and_choices": {
      "fixed_interval_nnz_direction_thread_blocking_operator_nnz_per_BMT
          ":
        [1, 2, 3, 4, 5, 9, 16, 20, 24],
      "grid_init_operator_block_size":
        [64, 128, 192, 256, 320]
    }
  }],
  "history": [{
    "node_id": 0, "template_id": "class_15",
    "values": [5, 256], "status": "ok", "gflops": 416.716
  }],
  "best": {
    "node_id": 0, "template_id": "class_15",
    "values": [5, 256], "gflops": 416.716
  }
}
```

Listing 3: Abridged generated SpMV kernel

```
// Excerpt from kernel_0; metadata setup is omitted.
unsigned int nzIndex = warpNzBase + (threadMetaId - threadMetaBegin);
```

```
for (unsigned int nzOffset = 0; nzOffset < valuesPerThreadMeta; nzOffset++)
{
   unsigned int processedNzColIndex;
   unsigned int nzColIndex;
   nzColIndex = nzColumnIndices[nzIndex];
   processedNzColIndex = nzColIndex;
   float nzValue;
   nzValue = nzValues[nzIndex];
   unsigned char rowStartFlag = bitmapFlags & 1;
   if (rowStartFlag == 1)
   {
      if (boundarySeen != 1)
      {
         headPartialSum = rowPartialSum;
      }
      else
      {
         y[currentRowIndex] = rowPartialSum;
      }
      rowIndexCursor = nzOffset == 0 ? rowIndexCursor : rowIndexCursor + 1;
      currentRowIndex = rowIndexCursor;
   }
   boundarySeen = boundarySeen | rowStartFlag;
   bitmapFlags = bitmapFlags >> 1;
   rowPartialSum = rowStartFlag?0 : rowPartialSum;
   rowPartialSum = rowPartialSum + nzValue * __ldg(&(x[processedNzColIndex]));
   nzIndex += (threadMetaEnd - threadMetaBegin);
   firstRowIndexForWriteback = processedFirstRowIndex;
}
headPartialSum = boundarySeen ? headPartialSum : rowPartialSum;
tailPartialSum = rowPartialSum;

// Warp-level segmented reduction; shuffle-up steps 2, 4, and 8 omitted.
warpFirstHeadSum = headPartialSum;
headPartialSum = __shfl_down_sync(0xFFFFFFFF, headPartialSum, 1);
headPartialSum = (laneId == 31) ? 0 : headPartialSum;
segmentPrefixSum = headPartialSum;
shuffledValue = __shfl_up_sync(0xFFFFFFFF, segmentPrefixSum, 1);
segmentPrefixSum = (laneId >= 1) ? shuffledValue + segmentPrefixSum : segmentPrefixSum;
// ...
shuffledValue = __shfl_up_sync(0xFFFFFFFF, segmentPrefixSum, 16);
segmentPrefixSum = (laneId >= 16) ? shuffledValue + segmentPrefixSum : segmentPrefixSum;
shuffledValue = __shfl_down_sync(0xFFFFFFFF, segmentPrefixSum, segmentOffset);
headPartialSum = shuffledValue - segmentPrefixSum + headPartialSum;
tailPartialSum = tailPartialSum + headPartialSum;

if (boundarySeen == true)
{
   if (laneId + segmentOffset == 31)
   {
      atomicAdd(&(y[currentRowIndex]), tailPartialSum);
   }
   else
   {
      y[currentRowIndex] = tailPartialSum;
   }
}
else
```

```
{
    warpFirstHeadSum += headPartialSum;
}
if (laneId == 0)
{
    atomicAdd(&(y[firstRowIndexForWriteback]), warpFirstHeadSum);
}
// Host-side data loading, benchmarking, and validation omitted.
```