SparseDitto: Customizing GPU Kernels for Different Sparsity Patterns with LLM-Based Agentic System

Shiyang Li
li004074@umn.edu
University of Minnesota
Minneapolis, MN, USA

Guangyan Sun
sun01158@umn.edu
University of Minnesota
Minneapolis, MN, USA

Jinwei Tang
tang0940@umn.edu
University of Minnesota
Minneapolis, MN, USA

Yanzhi Wang
yanz.wang@northeastern.edu
Northeastern University
Boston, MA, USA

Mingyi Hong
mhong@umn.edu
University of Minnesota
Minneapolis, MN, USA

Caiwen Ding
dingc@umn.edu
University of Minnesota
Minneapolis, MN, USA

Abstract

Sparse matrix kernels are fundamental to scientific computing, graph analytics, and machine learning. Their GPU performance depends strongly on the input sparsity pattern and execution strategy. For the same SpMM on the same matrix, cuSPARSE exhibits a 350× performance gap between CSR and Blocked-ELL. Our study of multiple data formats, specialized systems, and sparse compilers shows that no single implementation consistently dominates across sparsity patterns and operators. This motivates a system that can adapt its representation, execution strategy, and hardware mapping to each workload and target GPU.

We present SparseDitto, an LLM-based system that constructs a GPU kernel for each matrix, operator, and target GPU. SparseDitto supports SpMV, SpMM, and SpGEMM within a unified design framework. A lightweight additive model ranks established strategies using structural features of the input matrix. An architecture-aware planner then proposes several candidate designs. Coding and verification agents implement and refine them using measurements from the target GPU. Across three sparse operators and a diverse set of matrices, SparseDitto achieves a geometric-mean speedup of 2.68× over cuSPARSE on an NVIDIA RTX PRO 6000 GPU, with a maximum of 146.61×. On an NVIDIA H200 GPU, it achieves 2.79×, with a maximum of 78.5×. Its generated SpMM kernels also accelerate full-batch GCN training by up to 3.39×.

1 Introduction

Sparse matrix operations are core building blocks in scientific computing, graph analytics, and machine learning [14, 18, 26, 30, 31]. Their performance on GPUs can change by orders of magnitude even when the matrix, operator, library, and GPU remain unchanged. For one matrix with 355 K nonzeros, cuSPARSE completes SpMM in 37 μs using CSR but requires 13 ms using Blocked-ELL—a 350× difference caused solely by the representation and its associated kernel (§2.2). Although the blocked layout exposes a highly efficient arithmetic path, padding turns most of its memory traffic into useless work. This example illustrates a fundamental challenge in sparse GPU computing: a design that is highly effective for one sparsity pattern can be significantly inefficient for another.

Across sparse matrix–vector multiplication (SpMV), sparse matrix–dense matrix multiplication (SpMM), and sparse matrix–sparse matrix multiplication (SpGEMM), performance is determined not only by matrix dimensions, but also by the positions of nonzeros. Row-length distributions shape load balance [4]; column indices determine memory locality [29]; block occupancy controls whether blocked execution is worthwhile [31]; and, for SpGEMM, the distribution of intermediate products governs sparse-accumulation cost [13, 21, 33]. The storage format representation determines the layout of values and metadata, while the execution schedule determines workload partitioning, thread mapping, data movement, and hardware mapping (e.g., tile sizes and on-chip resource allocation) [7, 28, 29]. The two require joint optimization to achieve a desired GPU performance.

Prior work has developed effective optimizations for different regions of this design space. SELL [15] and CSR5 [18] improve regularity or load balance, while merge-based kernels distribute nonzeros more evenly [19]. CB-SpMV improves cache locality through adaptive blocking [4]. Tensor-Core SpMM kernels combine blocked representations with specialized dataflows [7, 31]. SpGEMM systems optimize row partitioning and sparse accumulation [13, 20, 21, 25, 33]. Their efficiency, however, comes from assumptions about the input structure, operator, accumulator behavior, or target architecture. These assumptions yield high performance when they hold. However, unmatched design can create severe performance cliffs when assumptions do not hold.

We obtain two key observations on four recent specialized or compiler-based systems: CB-SpMV [4], DTC-SpMM [7],

HSMU-SpGEMM [33], and SparseTIR [34] on an NVIDIA RTX PRO 6000 GPU. **First,** sparse optimization is highly asymmetric: a compatible representation and schedule can provide meaningful gains, whereas an incompatible pairing can incur orders-of-magnitude slowdowns. The best alternative format improves SpMV by up to 1.47× and SpMM by up to 2.8×, while a poor pairing loses up to 350×. Avoiding a mismatched design is therefore essential, but format selection alone is not sufficient. **Second,** no fixed implementation consistently dominates across sparsity patterns, operators, and GPUs. CB-SpMV is effective on cache-friendly blocked inputs; DTC-SpMM benefits matrices that retain sufficient useful work in compressed Tensor-Core blocks; and HSMU-SpGEMM performs well when its hash accumulator matches the intermediate-product distribution. Their relative performance changes with the GPU architecture and CUDA software stack. Such a fixed system therefore freezes two assumptions at the same time: the sparsity patterns it targets and the architecture on which it was tuned. These observations motivate the automatic construction of sparse GPU kernels, beyond sparse tensor compilers [6, 14, 34].

Recent LLM-based systems offer a complementary mechanism for open-ended code generation and optimization [17, 24, 32, 36], but unconstrained generation is inefficient: many functionally correct sparse kernels are poorly matched to a particular sparsity pattern or GPU. Effective automation therefore needs both *structured guidance* to focus generation and *target-GPU measurement* to resolve architecture-dependent uncertainty. In this paper, we present **SparseDitto**, an LLM-based agentic system that combines structural analysis, interpretable strategy ranking, architecture-aware planning, code generation, correctness validation, and target-GPU profiling. SparseDitto first introduces a unified design framework covering SpMV, SpMM, and SpGEMM together. This framework spans data representation and metadata, workload partitioning, dataflow, accumulation, and hardware-specific configuration. SparseDitto then instantiates this framework in an LLM-based agentic system that automatically search sparse kernel design. It can iteratively customize and optimize sparse matrix GPU kernels with architecture-aware planning and real hardware profiling feedback. In summary, this paper makes the following contributions:

- **A unified design framework for sparse GPU kernels.** We formulate sparse-kernel construction as the joint design of data representation and metadata, workload partitioning, thread mapping, dataflow, accumulation, and hardware-specific configuration. The framework covers SpMV, SpMM, and SpGEMM, and our cross-operator characterization demonstrates why these choices must be coordinated for each sparsity pattern.
- **Structure-guided and architecture-aware design space exploration.** A structural profiler computes 36 features that characterize the sparse pattern. An interpretable additive energy model ranks established optimization strategies from an offline measurement corpus, so the search starts near designs that already work on matrices with this structure. An architecture-aware planner agent combines that ranking with target-GPU constraints and instantiates several candidates.
- **End-to-end automated generation and optimization.** A coding agent realizes each plan in CUDA, and a verification agent checks correctness and profiles the candidate on the target GPU. Runtime measurement and profiling feedback then drive iterative refinement, so measurement on the target GPU decides which implementation is kept.

We evaluate SparseDitto on 60 SuiteSparse matrices, covering SpMV, SpMM at four dense widths, and SpGEMM. The results show that SparseDitto is faster than cuSPARSE in 94% of the evaluated cases, reaching a geometric mean of 2.68× on an RTX PRO 6000 (up to 146.61×) and 2.79× on an H200(up to 78.5×). The generated SpMM kernel also accelerates real GCN training, up to 3.39×.

2 Background and Motivation

GPU sparse kernels combine decisions in data representation, metadata management, execution schedule, and hardware mapping. The representation specifies the value layout and sparse metadata. The schedule specifies workload partitioning, thread mapping, and dataflow. Hardware mapping then selects tile sizes and resource allocation for the target GPU. Together, these choices determine memory traffic and available parallelism. They also determine synchronization cost and hardware utilization.

2.1 Sparse Matrix Kernels on GPUs

The sparsity pattern determines the distribution of nonzeros, the memory a kernel traverses, and the workload assigned to computing units. It also controls locality once the schedule is mapped to the GPU. The operator determines the work produced by each visited nonzero. As a result, the critical design choice changes across SpMV, SpMM, and SpGEMM.

SpMV computes $y = Ax$ and performs only two floating-point operations per nonzero. Its performance often depends on sparse metadata traffic and the locality of gathers from x. Load imbalance among parallel units is another major cost [8, 18, 19]. Row-based CSR avoids output reductions but inherits the row-length distribution. ELL and SELL use padding to regularize sparse traversal.

SpMM computes $C = AB$ with a dense matrix B of width K. Each nonzero of A drives K multiply–adds, which creates another pattern dimension and more reuse of B. The sparse representation needs to be coordinated with the mapping of rows or blocks to the GPU. The K-tile and output-accumulator placement also affect the dataflow. Dense sparse

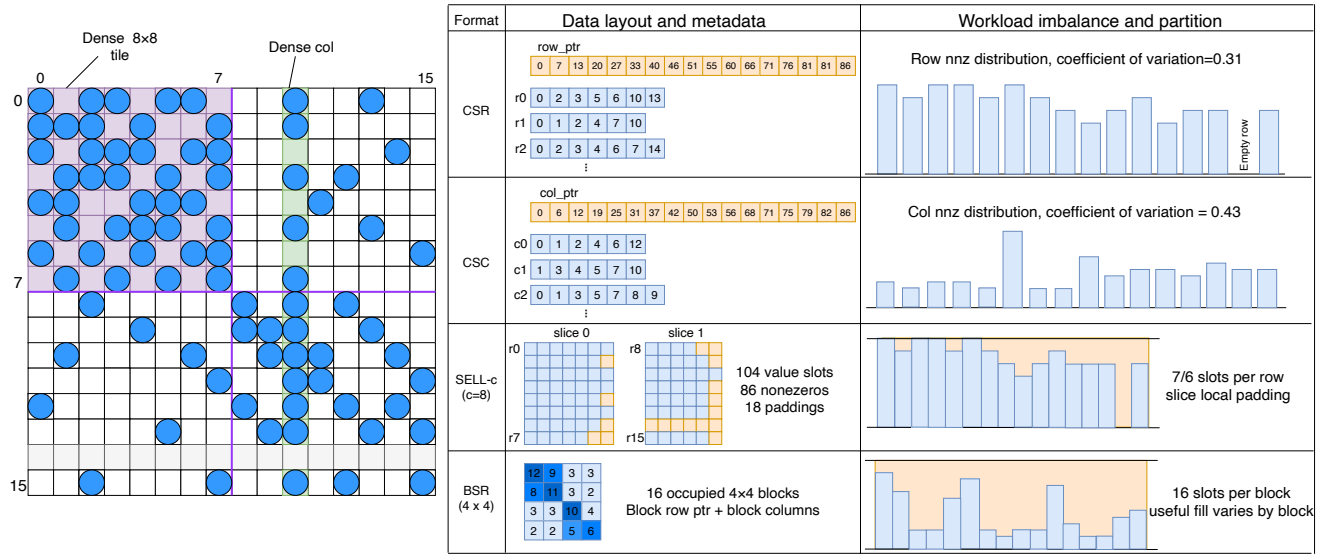

Figure 1. One 16×16 matrix exposes intrinsic pattern cues and format-induced costs. The matrix contains an irregular dense tile, a high-degree column, an empty row, and uneven row lengths. CSR and CSC traverse compressed rows and columns, respectively. SELL-C with $C = 8$ introduces slice-local padding, while 4×4 BSR introduces block fill. The right column shows the resulting work granularity and imbalance for rows, columns, slice slots, and blocks.

tiles can enable Tensor Cores when useful arithmetic outweighs compaction and fill-in costs [7, 31].

SpGEMM further discovers a sparse output. It computes $C = AB$ for two sparse matrices inputs A and B. In a row-wise algorithm, row i generates

$$u_B(i) = \sum_{k \in A[i,:]} nnz(B[k,:]), \tag{1}$$

The distribution of $u_B(i)$ affects row partitioning and temporary storage. It also determines the suitability of dense, hash, merge, or hybrid accumulators [20, 29, 33]. Design space now extends beyond input traversal because the kernel must construct output indices and intermediate state.

Thus, the same pattern can produce different metadata streams, processing units, and padding or fill costs. Figure 1 shows these effects on a common matrix. They are coupled to the parallel decomposition selected by the GPU kernel.

Two questions follow. First, when the GPU are fixed and only the data representation and its kernel change, how much does the sparsity pattern impact performance? (answered in §2.2) Second, the specialized systems that target part of whole design space: do they really cover what cuSPARSE leaves behind? (answered in §2.3)

2.2 Sensitivity to the Sparsity Pattern

We first hold the library and platform fixed to cuSPARSE [22]. It also varies the representation and SpMM width. The top panel of Figure 2 uses format names for brevity, but every bar measures a complete implementation.

The observed spread follows the assumptions of each implementation. SELL-C wins by up to 1.47× on a stencil with only 0.08% slice padding. It loses 33× when uneven row lengths inflate storage by 30×. BSR wins 1.25× when occupied 16×16 tiles are 87% dense. It falls to 0.04× on power-law graphs because the occupied blocks contain mostly zeros. COO pays for explicit row metadata and reductions on regular inputs. Under row imbalance, its nonzero-level decomposition makes it the most robust SpMM choice. COO wins 21 of 30 configurations and reaches 2.8×. Across the full sweep, the best alternative gains at most 2.8× over CSR. A mismatched design loses up to 350×.

The operator context changes the same tradeoff. At K=32, block-dense matrices can favor CUDA-Core blocked kernels. At K=128, Tensor-Core layouts replace them on some inputs. Blocked-ELL accelerates dense-tile arithmetic but pads every block row to a common width. This padding can increase transferred bytes by two orders of magnitude. For SpGEMM, the corresponding choices appear in row binning and phase structure. The accumulator dataflow also depends on the intermediate-work distribution [20, 21, 25, 33].

Observation 1. A sparse implementation couples assumptions about data layout and metadata traversal with assumptions about parallel decomposition and dataflow. A good match offers a bounded gain. A poor match can produce a much larger performance loss.

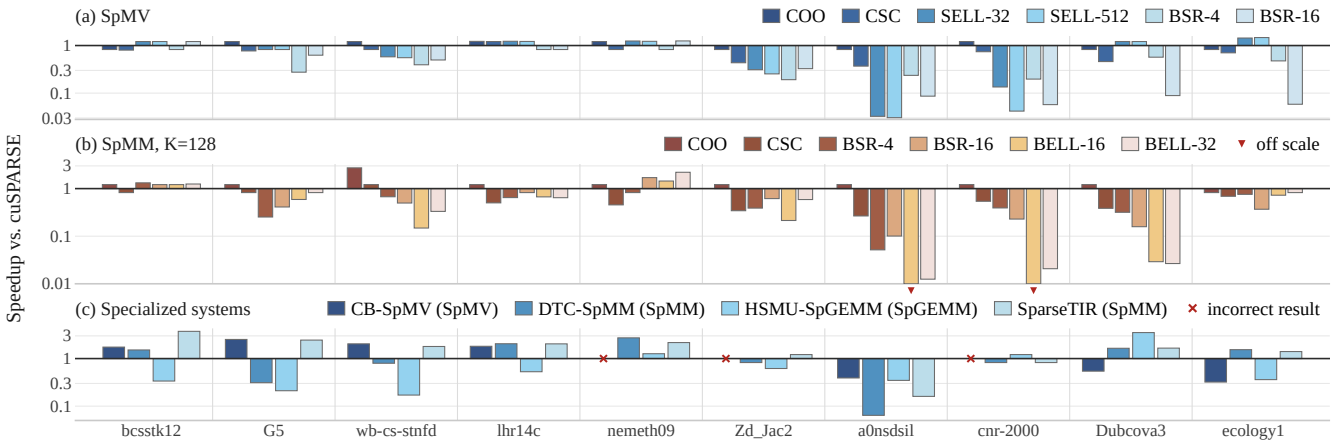

Figure 2. Sparse kernel performance sensitivity on ten SuiteSparse matrices [5]. Top: cuSPARSE implementations for different sparse representations in SpMV and SpMM at K=128. Bottom: four specialized systems. The best design changes with the pattern, while mismatched choices create severe slowdowns. All measurements use an NVIDIA RTX PRO 6000 Blackwell GPU with CUDA 13.0 and report the mean of ten timed iterations after warmup.

2.3 Limitations of Specialized Systems

We next port one recent state-of-the-art specialized system per operator to the same NVIDIA RTX PRO 6000 GPU platform. CB-SpMV [4] combines cache-friendly blocks with adaptive per-block formats. DTC-SpMM [7] combines compressed block metadata with Tensor-Core execution. HSMU-SpGEMM [33] uses a shared-memory hash accumulator for SpGEMM. SparseTIR [34] coordinates dynamic data format scheduling for SpMM as compiler.

Figure 2 shows how they perform. CB-SpMV achieves 1.7–2.5× while its blocked working set remains cache-friendly. It loses on every matrix above 3 M nonzeros and falls to 0.32×. DTC-SpMM reaches 3.1× on block-structured inputs. It also beats cuSPARSE's padded Tensor-Core format by 26–106× on irregular matrices. However, under extreme row imbalance DTC-SpMM falls to 0.04×. HSMU-SpGEMM surpasses cuSPARSE on three and loses on seven of ten matrices. None of the three systems dominates the full set.

The target platform changes the useful region further. These systems were developed for earlier GPU and CUDA generations. On Blackwell and CUDA 13.0, their fixed resource choices meet different hardware conditions, and they adapt poorly to the new platform. DTC-SpMM, for example, reports more than 1.5× over cuSPARSE on an RTX 4090 [7]. Running the released artifact on our Blackwell GPU with CUDA 13.0, we measure it below cuSPARSE at every dense width, from 0.73× at K=8 down to 0.58× at K=256.

Sparse tensor compilers expose a broader space through dynamic format decomposition. They also provide scheduling primitives. SparseTIR [34] beats cuSPARSE on 25 of 30 SpMM configurations. Its geometric-mean speedup is 1.40× and its maximum is 5.0×. SparseTIR nevertheless falls to 0.13–0.20× on the matrix behind the 350× format cliff because long rows serialize execution. SparseTIR can express a bucketed hybrid layout, but a developer supplies the format rewrite and schedule [34]. Its coverage therefore depends on the implementations encoded in its rule set. It also does not support SpMV or SpGEMM, and it covers only a limited set of widths (K = 32, 128, 256) for SpMM.

Observation 2. Specialized systems and sparse compilers provide effective implementation mechanisms. A system fixed to one operator or an enumerated rule set, still covers only part of the joint design space.

The measurements behind Observations 1 and 2 reveal a practical consequence. A user who runs different sparse operations on a GPU has to switch between frameworks to stay fast, picking a different one for each sparsity pattern and each operator. And the system that wins today can fail or slow down sharply on the next GPU.

Why can't we have a unified optimization system across various sparse patterns, operators, and hardware?

These observations lead to three requirements for such a system. First, a common design framework must describe both representation and execution choices. It should also capture their hardware mapping. Second, structural analysis can expose the condition that enables a better optimization strategy. Third, the system needs to combine measured experience with hardware-aware search.

3 SparseDitto Design

SparseDitto generates a sparse kernel for a matrix X, an operator o, and a target GPU H. The output is a validated CUDA implementation with its required data structures.

SparseDitto describes each candidate configuration as

$$\Pi = \langle R, S, \theta_H \rangle, \qquad R = \langle L, I \rangle, \quad S = \langle P, D \rangle. \tag{2}$$

The representation R contains a data layout L and sparse metadata I. The layout specifies grouping, reordering, padding, and value placement. The metadata defines how the kernel traverses the sparse matrix. The execution schedule S contains parallel decomposition P and dataflow D. The former maps the sparse iteration space to CTAs, warps, and threads. The latter controls operand movement and accumulation. Finally, θ_H contains target-specific parameters such as tile size and kernel launch geometry. It also specifies shared-memory usage and other hardware resource choices.

An optimization strategy identifies a reusable design family, such as SELL-based SpMV or BSR Tensor-Core SpMM. A candidate configuration binds the strategy to one $\langle X, o, H \rangle$. For example, a SELL configuration fixes the row ordering and slice boundaries. It also selects the slice metadata and thread mapping.

Figure 3 shows the overview of SparseDitto workflow. SparseDitto first extracts structural features from the sparse input (§3.1). An additive energy model ranks strategies from an offline measurement corpus (§3.2). The planner uses this ranking to seed a hierarchical search under the target-GPU constraints (§3.3). Code generation then realizes each candidate from tested sparse mechanisms. Correctness checks and target-GPU measurements select the promising candidates and guide further optimization (§3.4).

The following sections describe how SparseDitto selects and coordinates these choices for a matrix and target GPU, reasoning over 36 structural features. The supplementary material instantiates these features in detail.

3.1 Structural Features and Cost Estimates

The feature design starts from known sparse optimizations. For each mechanism, we identify the structural condition behind its speedup and the cost of violating that condition. We define result

$$\mathcal{E}(X, \Pi, o, H) = \langle E_{\text{intr}}, E_{\text{repr}}, E_{\text{op}}, E_{\text{hw}} \rangle. \tag{3}$$

E_{intr} describes the input without assuming a candidate. E_{repr} estimates costs introduced by R. E_{op} captures the operator-dependent intermediate work and accumulator state created by o. E_{hw} records constraints from the target GPU. Figure 1 illustrates the intrinsic pattern cues and representation-induced costs on a common matrix. The intrinsic features follow four familiar sparse-matrix properties: work distribution, locality, reuse, and tile structure.

Intrinsic pattern features. Let $r_i = nnz(X[i, :])$. Matrix size and density describe the total sparse working set. We record the average, maximum, and coefficient of variation (CV) of r_i. Quartiles and the empty-row share expose distribution details hidden by these aggregates. These features indicate the imbalance addressed by CSR5 and merge-based decomposition. They also indicate the potential benefit of SELL-C-σ [15, 18, 19].

Column-degree statistics describe reuse and asymmetry between the two matrix dimensions. Locality is represented by row span and its normalization by matrix width. The mean gap between sorted column indices and the diagonal incidence provide two additional measures. We also record occupied 16×16 tile coverage and the share of nonzeros inside those tiles. These quantities expose block structure relevant to Tensor-Core and adaptive-block designs [4, 7, 31]. Together, this group contains 18 continuous features. A categorical summary divides the row distribution into five classes: uniform, moderate, skewed, power-law, and bimodal.

Representation-induced features. The representation $R = \langle L, I \rangle$ changes the value slots and metadata traversed by a kernel. SparseDitto computes value-slot expansion for padded and blocked layouts:

$$\begin{aligned} \rho_{\text{ELL}} &= m \max_i r_i / nnz(X), \\ \rho_{\text{SELL}}(c) &= \textstyle\sum_s |s| \max_{i \in s} r_i / nnz(X), \\ \rho_{\text{BSR}}(b) &= b^2 nnzb_b / nnz(X), \end{aligned} \tag{4}$$

where s denotes a row slice and $nnzb_b$ counts occupied b×b blocks. These factors estimate padded computation and value traffic. The storage bound also accounts for slice offsets and block-column indices. Row permutations are included when required. SparseDitto evaluates ELL and three SELL slice heights. It also evaluates BSR with $b \in \{4, 8, 16\}$. These choices produce eight continuous representation features. Their values are recomputed after a change to L, I, or the format parameters.

Operator-induced features. Input statistics alone do not expose the intermediate iteration space of SpGEMM. From Equation 1, SparseDitto computes the total, maximum, p95, and CV of $u_B(i)$ for the requested product. It also computes the total and CV for $B = X^T$. The accumulator-fit statistic

$$F_h = \frac{1}{m} \sum_i \mathbf{1}[u_B(i) \le h] \tag{5}$$

reports the fraction of rows below capacities h = 512, 4K, and 32K. These thresholds distinguish accumulator regimes that a mean would hide. SparseDitto also estimates

$$\gamma_B = \textstyle\sum_i u_B(i) / nnz(C), \tag{6}$$

which measures the intermediate products merged into each output nonzero. The estimator uses stratified sampling inspired by MatRaptor's dataflow analysis [29]. The ten features in this group describe intermediate work and imbalance. They also expose accumulator demand and output compression.

Hardware and task context. E_{hw} contains physical capacities and legal kernel launch bounds. It records the SM count and cache capacity. It also records limits on threads, registers, and shared memory. The operator and SpMM width K specify the task. These values guide strategy selection and

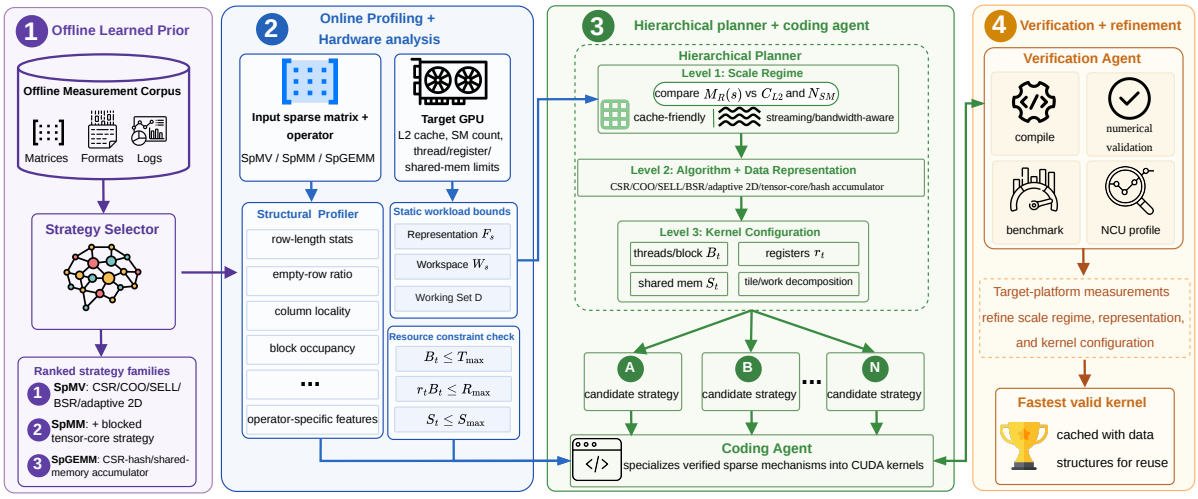

Figure 3. SparseDitto constructs and refines sparse GPU implementations. Structural features and an additive energy model rank known optimization strategies. Architecture-aware planning coordinates the representation with the execution schedule and hardware parameters. Compilation, validation, and target-GPU measurement refine each candidate.

planning, but they are not counted among the 36 continuous sparse features.

The feature vocabulary follows established sparse kernel optimization mechanisms. Row statistics come from load-balancing methods [4, 26, 28, 29]. Tile features come from blocked and Tensor-Core designs [6, 7, 31, 34, 35]. Format definitions [8, 11, 15, 18, 20, 22] provide the exact expansion formulas. SpGEMM systems motivate the accumulator features [13, 21, 33]. Corpus measurements determine useful parameter resolutions and capacity thresholds. SparseDitto computes all features from bounded scans and occupied-block counts. The compression estimate uses budgeted sampling, and no GPU kernel runs during this process.

3.2 Energy-Based Strategy Selection

The selector uses measured experience to rank established optimization strategies [4, 7, 22, 27, 33–35]. Its labels retain parameters with a large effect on format overhead or thread mapping. The planner later selects the remaining hardware-specific parameters. The selector vocabulary V_o is a restricted subset of that construction space. For SpMV, it contains CSR and COO. It also contains SELL with slice heights $\{16, 32, 512\}$, BSR with block sizes $\{4, 8, 16\}$, and adaptive 2D blocking. The SpMM vocabulary contains CSR, COO, the same three BSR variants, Blocked-ELL-$\{16, 32\}$, and the DTC Tensor-Core design. The three SpGEMM labels are CSR-hash, shared-memory tiled hash, and hybrid accumulation. These strategies draw on established GPU kernels [7, 15, 29].

The selector is a single additive energy model [12, 16]. Its input x concatenates the 36 standardized (log-scaled) sparsity features, a one-hot shape class, an operator indicator, and the standardized tile width $\log_2 K$ (nonzero only for SpMM). For each scalar coordinate x_j, a nonlinear basis *shared across operators* produces

$$h_j(x_j) = \tanh(x_j w_{1j} + b_{1j}) \in \mathbb{R}^{16}. \tag{7}$$

Each operator o then has its own readout for every strategy $s \in V_o$:

$$E_o(s \mid x) = \beta_{o,s} + \sum_j h_j(x_j)^\top w_{2,o,j,s}. \tag{8}$$

Lower energy indicates a stronger recommendation, and the recommendation is $\arg\min_{s \in V_o} E_o(s \mid x)$. Sharing the basis across operators lets a single feature response be estimated from all three tasks, while the operator-specific readouts capture the different effect of that feature on SpMV, SpMM, and SpGEMM. The nonlinear basis can express a threshold response, such as a sharp penalty once SELL padding exceeds a level, without a hand-designed cutoff.

We train the energy directly from measured runtimes. For a training example, let $M \subseteq V_o$ be the strategies actually benchmarked and $t_s = \log_2(\text{baseline runtime}/\text{runtime of } s)$ their measured log-speedups. The objective combines a listwise likelihood [3] with a gap-weighted pairwise margin [2]:

$$\mathcal{L}_o = \sum_{(x,\mathbf{t})} \Bigg[-\sum_{s \in M} q_s \log p_o(s \mid x) + \lambda_o \sum_{\substack{s,s' \in M \\ t_s > t_{s'}}} (t_s - t_{s'}) \operatorname{softplus}\big(E_o(s \mid x) - E_o(s' \mid x)\big) \Bigg] \tag{9}$$

where $p_o(s \mid x) = \operatorname{softmax}_{s \in M}\big(-E_o(s \mid x)\big)$ and $q_s = \operatorname{softmax}_{s \in M}(t_s/T)$ are soft targets from the measured speedups. The listwise term keeps near-optimal alternatives ranked highly; the margin term orders strategies by measured speed with a penalty proportional to the speedup gap. The operator-specific weight λ_o sets how strongly the margin acts: it is

small for SpMV, whose strategy speedups cluster tightly, and larger for SpMM and SpGEMM.

The additive form makes each score decompose into the per-feature terms $h_j(x_j)^\top w_{2,o,j,s}$, so the model is interpretable by construction [1]. The planner receives these terms as feature-level explanations of why a strategy was ranked where it was.

The selector returns a ranked set of strategies rather than a final implementation. The learned ranking therefore guides the known part of the design space, while target-GPU search determines the final implementation.

3.3 Hierarchical Architecture-Aware Planning

The planner receives the structural features and strategy ranking. It also receives the target GPU specification. From these inputs, it constructs several candidate configurations. Each candidate records R, S, and θ_H. Storage estimates and resource bounds accompany these choices.

Planning proceeds hierarchically to reduce the search space. The first level compares workload scale and available parallelism with physical hardware capacities. The second level chooses the algorithm, data representation, and execution schedule. The third level binds a legal kernel configuration to the target GPU.

Static workload bounds. Before code generation, SparseDitto computes quantities fixed by a candidate and the sparse input. Let

$$M_{\text{rep}}(\Pi) = M_{\text{value}}(L) + M_{\text{meta}}(I) \tag{10}$$

be the complete representation footprint. Let $W(\Pi)$ denote temporary storage introduced by the dataflow. The read-side and total working sets are

$$M_R(\Pi) = M_{\text{rep}}(\Pi) + W(\Pi) + D_R, \tag{11}$$

$$M_T(\Pi) = M_R(\Pi) + D_S, \tag{12}$$

where D_R and D_S are common input and output operands. For SpMM, these terms include dense B and C. For SpGEMM, $W(\Pi)$ reflects the selected accumulator and intermediate representation.

SparseDitto reports these byte counts against physical cache and memory capacities. It also compares the number of independent work units with the GPU's SM count. A work unit can be a row, slice, nonzero partition, or tile. These comparisons are static bounds rather than predictions of cache residency or utilization. Measurements after code generation determine the realized behavior.

Hardware resource bounds. For a kernel configuration t, let B_t be its thread count. Let r_t denote registers per thread and S_t denote shared memory. A legal configuration must satisfy

$$B_t \le T_{\max}, \qquad r_t B_t \le R_{\max}, \qquad S_t \le S_{\max}. \tag{13}$$

The planner checks the requested threads and shared memory before code generation. The compiler-reported register count completes the check after compilation. A hard-capacity violation removes the candidate. Cache and occupancy estimates remain guidance for comparing legal configurations.

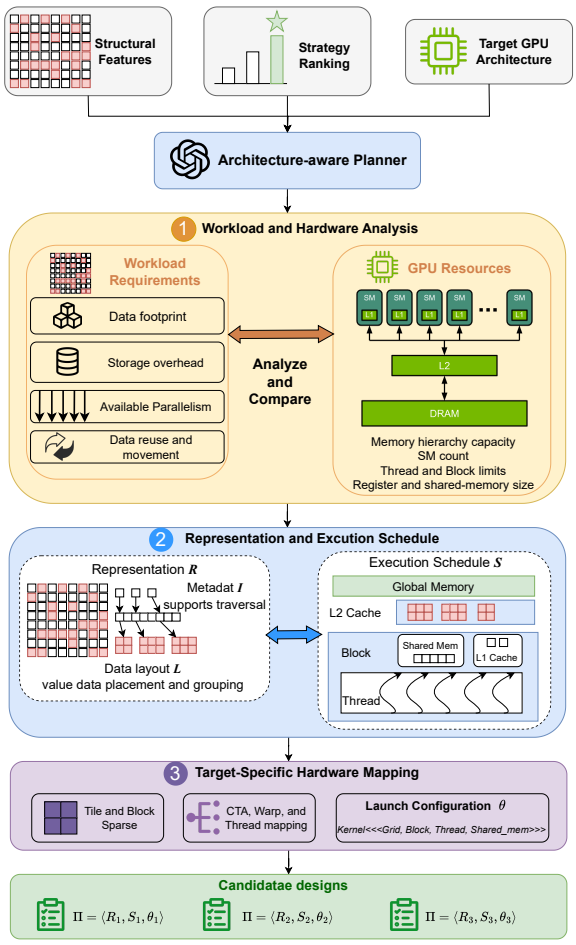

Figure 4. Hierarchical architecture-aware planning in SparseDitto. Several branches are measured on the target GPU, and feedback can revise choices at earlier levels.

Coordinating representation and execution. Within these bounds, the planner instantiates each strategy across all parts of Π. For example, a SELL candidate binds row slicing in L to slice offsets and an optional permutation in I. It maps slices through P and selects c, σ, and launch geometry in θ_H. A nonzero-balanced CSR candidate retains the value layout but adds partition boundaries to I. Its schedule partitions the nonzero stream and performs segmented reduction. A Tensor-Core SpMM candidate connects its block layout to block-tile assignment and dense-operand staging. A SpGEMM hash candidate connects row binning to accumulator dataflow and hash capacity.

The selector seeds candidates from high-ranked strategies. The planner then adds more alternatives with a different representation or workload partitioning strategy. Each candidate receives resource-legal parameters for H, followed by updated static bounds.

Each candidate starts an independent search branch. Initial iterations are distributed across the branches to preserve design diversity. Later iterations favor the fastest valid candidates while retaining the best result from each strategy family. This organization gives the language model room to be flexible and inventive.

3.4 Iterative Code Generation and Optimization

Code generation follows the selected representation and execution schedule. SparseDitto provides conversion tools and metadata builders for CSR, COO, ELL, SELL, and BSR. It also provides adaptive 16×16 blocks inspired by CB-SpMV [4]. Packed column windows for SpMM follow Sextans [28]. SpGEMM support includes upper-bound analysis and row binning. Scans and workspace management follow the row-wise structure of MatRaptor [29].

These mechanisms implement L and I while providing building blocks for P and D. The coding agent applies the selected thread mapping and operand staging. It also implements the accumulator and θ_H parameters. Generated kernels use a common harness and must reproduce the complete sparse operation.

Every generated implementation is compiled and numerically validated. SparseDitto then measures latency and profiles individual CUDA kernels with NCU on the target GPU. The resulting diagnosis updates a specific part of Π. Excessive padding or metadata traffic triggers a change to L or I. Imbalance and synchronization overhead trigger a change to P. Cache behavior and workspace traffic guide changes to D. Resource pressure changes θ_H. The planner updates the candidate configuration before the coding agent revises the implementation. This process keeps the representation and execution schedule consistent across iterations.

SparseDitto returns the fastest numerically valid candidate across all branches. The cached result is identified by its sparsity pattern, operator, and target platform. For SpMM, the key also records K.

4 Evaluation

Workloads. We use a stratified set of 60 matrices from the SuiteSparse Matrix Collection [5]. It covers regular stencils, finite-element matrices, circuit and process-simulation problems, power-law web graphs, road networks, and gene networks, and includes the hardest cases from the evaluations of the specialized systems we compare against. For each matrix, we evaluate SpMV, SpMM with $K \in \{8, 32, 128, 256\}$, and SpGEMM. For SpGEMM, we follow [33], computing AA for square matrices and AA^{T} otherwise.

The evaluation set covers various sparse matrices from SuiteSparse [5]. Rows span 359–16,777,216. NNZ ranges from 817 to 101 million, at densities from 1.78×10^{-7} to 0.110. Mean row length ranges from 1.0 to 1,107.1, with coefficient of variation spans 0–12.2.

Platform and baselines. We evaluate on two GPU architectures, an NVIDIA RTX PRO 6000 Blackwell with CUDA 13.0 and an NVIDIA H200 Hopper with CUDA 12.9. We report the main result on both. The ablation study and the cache and memory-traffic analysis run on the RTX PRO 6000. We use the corresponding cuSPARSE implementation [22] as the per-task reference. We benchmark all applicable cuSPARSE algorithms under the same numerical contract and use the fastest valid configuration as the baseline. For SpMV, we compare with CB-SpMV [4]. We also try to port AlphaSparse [6], a generator of machine-designed SpMV formats and kernels. Its released implementation fails on more than half of tested matrices. For SpMM, we compare with DTC-SpMM [7] and SparseTIR [34]. For SpGEMM, we compare with HSMU-SpGEMM [33]. Each comparison uses the input configurations supported by the released artifact, except that we extend DTC-SpMM to support $K = 8$.

LLM configuration. All three agents, the planner, coder, and verifier, call the same model: GPT-5.6-terra [23] with reasoning effort *low* and a budget of 8,196 tokens per call. All other parameters are left at the API default. Every task runs the same search budget: three strategy branches with up to five generate-compile-benchmark iterations each, then five more for the two fastest branches.

Measurement protocol. We time the generated kernel and baselines with CUDA events. Each implementation has five warmup executions followed by ten timed executions, and we report their arithmetic mean, excluding the one-time preprocessing. Aggregate speedups are geometric means over tasks, reported with a 95% confidence interval obtained by bootstrapping the tasks (20,000 resamples of the log-speedups).

Correctness validation. We use the corresponding FP32 cuSPARSE implementation as per-task correctness reference. SparseDitto and cuSPARSE receive identical input values. For SpMV and SpMM, a candidate is considered correct only if every output entry satisfies

$$\left|\widehat{y}_i - y_i^{\text{ref}}\right| \le \epsilon_{\text{abs}} + \epsilon_{\text{rel}} \left|y_i^{\text{ref}}\right|, \tag{14}$$

where $\epsilon_{\text{abs}} = 10^{-5}$ and $\epsilon_{\text{rel}} = 10^{-3}$. Any runtime error, unexpected `NaN` or infinity, or violation of this bound causes the candidate to fail validation.

For SpGEMM, correctness includes both the sparse output structure and the associated values. We canonicalize the SparseDitto and cuSPARSE outputs by sorting column indices within each row and coalescing duplicate entries. The output dimensions, number of nonzeros, row pointers, and column indices must match exactly. Every corresponding nonzero value must additionally satisfy the same mixed absolute and relative error bound above. We apply this single predeclared validation rule uniformly to all matrices, generated candidates, and comparison systems.

4.1 Generated Kernel Performance

We organize the results by operator and report each one on both GPUs. On seven of the largest matrices the cuSPARSE SpGEMM implementation triggers an out-of-resources error, so those tasks have no reference and are excluded. Each additional system is evaluated only for the operator and configurations supported by its released artifact. All speedups below are geometric means over the validated tasks. On the RTX PRO 6000, SparseDitto reaches 2.68× over cuSPARSE (95% CI [2.48, 2.89]). On the H200 the pipeline runs unchanged: it reads the device properties of the new target and regenerates every kernel from scratch. It reaches 2.79× (95% CI [2.61, 2.99]), an interval that overlaps the first, so the aggregate result does not separate the two architectures. Figures 5 and 6 separate both platforms by operator and dense width.

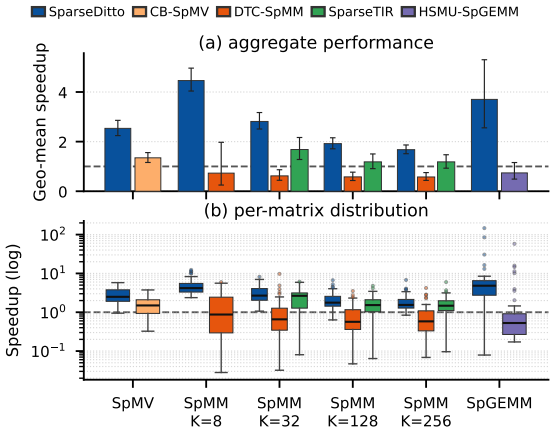

Figure 5. Generated kernel performance for SpMV, SpMM, and SpGEMM. (a) geometric-mean speedup over cuSPARSE; whiskers are 95% bootstrap confidence intervals over tasks. (b) the per-matrix distribution on a logarithmic scale. The dashed line is cuSPARSE.

SpMV. Against cuSPARSE, SparseDitto achieves 2.53× on the RTX PRO 6000 and 2.68× on the H200. CB-SpMV achieves 1.35× and 1.70× on the two GPUs. On the same inputs, SparseDitto is 1.91× and 1.57× faster than CB-SpMV.

AlphaSparse generates a customized SpMV implementation for each matrix by searching under a fixed static rule set. However, under our validation rule it produces a correct kernel on only 13 matrices, under a 9 hours search budget per matrix. The remaining runs crash or return values outside the bound. On the 13 matrices it handles correctly, SparseDitto is 5.34× faster, and AlphaSparse only reaches 0.62× over cuSPARSE.

SpMM. Against cuSPARSE, SparseDitto achieves 4.46× at $K = 8$, 2.81× at $K = 32$, 1.92× at $K = 128$, and 1.68× at $K = 256$. The gain is largest at the small dense widths and narrows as K grows. The H200 follows the same trend, at 4.18×, 2.81×, 1.96×, and 1.87×.

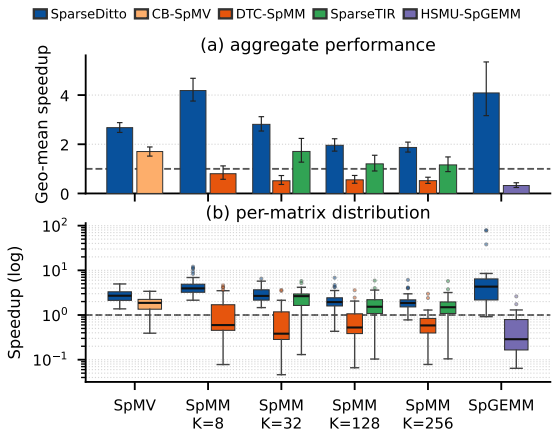

Figure 6. The same measurement on the H200. Both panels follow Figure 5.

For SpMM, we also compare with SparseTIR and DTC-SpMM. SparseTIR reaches 1.68× at $K = 32$. Its speedup is 1.18× at $K = 128$ and 1.18× at $K = 256$. DTC-SpMM achieves 0.73× at $K = 8$ and 0.62×, 0.58×, and 0.58× at the three larger k, so it stays below cuSPARSE at every width. Across the evaluated tasks, SparseDitto is 1.57× faster than SparseTIR and 3.99× faster than DTC-SpMM. Both margins hold on the H200, at 1.63× and 4.32×.

SpGEMM. SparseDitto achieves 3.70× over cuSPARSE on the RTX PRO 6000 and 4.09× on the H200. And its maximum is 146.61×. The wider distribution reflects variation in intermediate work. HSMU-SpGEMM achieves 0.74× on the square matrices it supports, and drops to 0.32× on the H200. SparseDitto is 5.37× faster on these matched inputs and 13.60× faster on the H200, and it covers the rectangular inputs that HSMU-SpGEMM does not support.

Panel (b) of both figures reports the per-matrix distribution. At $K = 8$ and $K = 32$ the whole SpMM distribution sits above cuSPARSE on both GPUs, so the mean is not carried by a few favorable inputs. As K grows the box contracts toward the baseline, from an interquartile range of 3.28–5.57× at $K = 8$ to 1.27–2.22× at $K = 256$ on the RTX PRO 6000. SpGEMM has by far the widest spread on both platforms, with an upper tail at 146.61× and 78.5×, because the intermediate work varies much more across matrices than the streaming work of SpMV and SpMM.

4.2 Cache and Memory-Traffic Analysis

To explain why the generated kernels are faster, we profile the timed every winning kernel, its cuSPARSE reference, and each specialized system with NVIDIA Nsight Compute on the same inputs. The profiler replays each launch with flushed caches and locked base clocks, so the counters reflect

Comparison	ΔL1	L1 req.	ΔL2	L2 req.	Occ.	BW
vs. cuSPARSE	+4.7	1.09×	+3.6	0.77×	1.96×	1.98×
vs. CB-SpMV	+1.6	0.63×	−7.3	0.57×	0.93×	1.03×
vs. SparseTIR	−18.6	0.85×	−2.2	1.03×	1.24×	1.80×
vs. DTC-SpMM	−5.1	1.44×	−1.3	1.13×	3.21×	5.03×
vs. HSMU-SpGEMM	−16.3	0.83×	+1.5	0.97×	0.88×	4.25×

Table 1. SparseDitto against each system on evaluated tasks. ΔL1 and ΔL2 are our hit rate minus theirs, in percentage points. The remaining columns are ours divided by theirs: L1 and L2 sector requests per operator invocation, achieved occupancy (active warps as a share of the hardware maximum), and achieved DRAM bandwidth.

each kernel's own access behavior. When a compute phase launches several kernels, we pool the sector counters over all launches. We report the cache hit rates, the number of cache sector requests, and the DRAM traffic. Traffic means the bytes read and written at device memory. Requests and traffic are normalized to one operator invocation: one SpMV, one SpMM, or one SpGEMM.

Versus cuSPARSE. Table 1 reports every comparison. Against cuSPARSE the generated kernels issue slightly more L1 requests but fewer L2 requests, so a warmer L1 absorbs traffic that would otherwise reach L2 and then memory. And generated kernels keep about twice as many warps resident, which sustains about twice the DRAM bandwidth. The gain is concentrated where reuse exists. On SpMM the median task raises L1 by 8 to 17 points depending on the width; on SpGEMM it shows up in L2 instead, because the chosen formats keep intermediate products on chip. SpMV is the exception: hit rates stay within a few points of cuSPARSE, so its wins come from balancing work across rows rather than from locality. This matches Figure 7, where the maximum row length governs SpMV.

Versus specialized systems. SparseTIR and HSMU-SpGEMM reach L1 hit rates well above ours, yet SparseDitto is 1.57× and 5.37× faster on the same inputs. A hit rate is a ratio, not a volume: it reports how often the requests a kernel issues hit, not whether those requests, or the work behind them, are necessary. Both systems re-read staged data, and those re-reads hit without reducing work. The last two columns show what decides the runtime instead. Against SparseTIR we issue fewer requests at both levels. HSMU-SpGEMM keeps as many warps resident as we do, yet sustains a quarter of our bandwidth: its pipeline issues 24 kernels per SpGEMM against our 5, and each boundary costs a kernel launch and a device-wide synchronization that cannot hide by overlap. CB-SpMV re-reads its blocked data the same way, and we issue 0.63× L1 requests for the same product.

DTC-SpMM is the case that looks contradictory: its hit rates are level with ours and it moves slightly less data, yet we are 3.99× faster. Moving less data only helps if memory is the limit, and for DTC-SpMM nothing is. It keeps a third of our resident warps and sustains a fifth of our bandwidth, and a direct probe puts its Tensor-Core pipeline at 1.5% of peak. Its 16×8 tiles have a median occupancy of 10.1% on these matrices, so each warp it launches carries almost no useful work: too little to feed the Tensor Cores it exists to use, and too few warps to hide the latency of fetching them. The effect tracks the padding ratio, from 6.40× at K=8, where the tile is emptiest, to 2.89× at K=256. Across all four systems the runtime follows the work actually issued and the rate it is issued at, while the hit rate follows the chosen format.

Configuration	SpMV	SpMM	SpGEMM
Task only	1.93×	2.02×	1.22×
Pattern + Task	2.27×	2.28×	2.24×
SparseDitto	**2.53×**	**2.52×**	**3.70×**

Table 2. Ablation results by operator. Each cell reports the average speedup over cuSPARSE.

4.3 Ablation Study

To separate the contributions of pattern analysis and architecture-aware planning, we evaluate three configurations under the same generation budget and tree-search setting. *Task only* receives the task description as its only input. *Pattern + Task* adds sparsity-pattern analysis. The full configuration further adds architecture-aware planning. All three configuration have the selector giving strategy suggestion.

Table 2 reports the result of each configuration for the three operators. With the task description alone, SparseDitto already beats cuSPARSE. The pattern analysis can further improve achieved performance, raising SpMV from 1.93× to 2.27×, SpMM from 2.02× to 2.28×, and SpGEMM from 1.22× to 2.24×. Adding architecture-aware planning gives the full system reported above. This consistent progression shows that pattern analysis and architecture-aware planning both help the model generate better sparse kernels.

4.4 Strategy Selector

We next evaluate the energy model in isolation: how well it ranks existing strategies, whether the energy formulation helps over a plain classifier, and which structural features drive its decisions.

Training cost. Fitting the selector is a negligible one-time cost. The model has only 15.9K parameters, and training on the full corpus of 400 matrices for 600 epochs takes about two seconds on a single RTX PRO 6000 GPU or H200 GPU. Because training is offline, its cost is amortized over every matrix and kernel the selector subsequently guides, and extending the corpus with new measurements retrains it at the same negligible cost. At inference the model ranks strategies

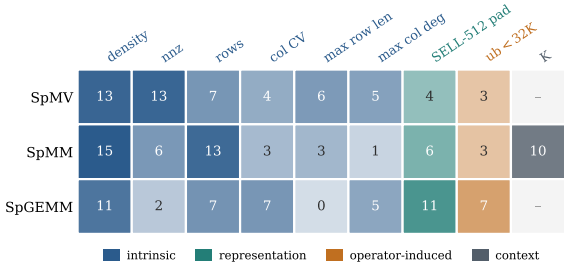

Figure 7. Structural feature importance per operator. Each cell gives a feature's share of permutation importance in percent; colors group the feature families, and a dash marks a feature the operator does not use.

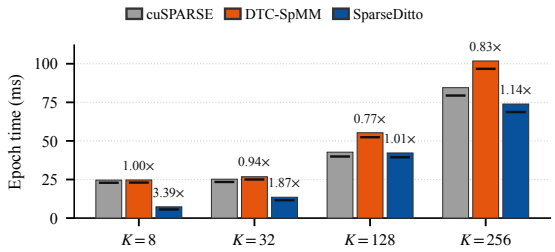

Figure 8. Full-batch GCN epoch time on Reddit across hidden widths. Horizontal marks inside bars denote SpMM time. Labels above DTC-SpMM and SparseDitto report speedup over cuSPARSE.

in well under a microsecond, negligible beside feature extraction and the downstream code generation, so the selector adds no measurable overhead to the generation pipeline.

Ranking quality. We rank the strategies using an offline corpus of 400 SuiteSparse matrices with measured runtimes for every strategy. There is no overlap between this training set and our test set for main results in Section 4.1. We collect strategies from many prior open-source works [4, 7, 18, 26–29, 31, 33]. The energy model outperforms a multilayer perceptron trained with the same listwise objective on all three operators. The proposed interpretable model also remains competitive with gradient-boosted trees [9]. Full protocols, metric definitions, and per-operator results are provided in the supplementary material.

Which features matter. Because the energy is additive, each recommendation decomposes into per-feature energy contributions, giving a direct, model-native read on which structural features drive the ranking (Figure 7). Rather than hinging on a single feature, the model draws on a broad set of correlated structural signals. Features like matrix density, nonzero count, and row count are consistently influential across all three operators, with operator-specific signals layered on top: the dense width K for SpMM, and accumulator-demand statistics (intermediate-product and AA^{T} upper bounds) for SpGEMM. The full per-feature breakdown is given in the supplementary material. All features are computed from bounded scans at negligible cost, and which of them matter is input and operator-specific.

4.5 End-to-End Application Evaluation

Following DTC-SpMM's end-to-end protocol [7], we evaluate the generated kernels in a two-layer full-batch GCN on Reddit [10]. We use an unweighted symmetric adjacency and sweep $K \in \{8, 32, 128, 256\}$. Each result averages 100 epochs after 10 warmup epochs. Each epoch invokes four sparse aggregations: two in the forward pass and two in the backward pass. Symmetry lets the backward pass reuse the corresponding forward kernel. The generated implementation replaces only these aggregation kernels.

Figure 8 reports the result. At $K = 8$, SparseDitto reduces the average epoch time from 24.6 to 7.3 ms, yielding a 3.39× end-to-end speedup over cuSPARSE. At $K = 32$, it reduces the epoch from 25.2 to 13.5 ms and achieves 1.87×. The advantage narrows as the dense dimension grows, but SparseDitto remains faster than cuSPARSE. It reaches 1.01× at $K = 128$ and 1.14× at $K = 256$. DTC-SpMM matches cuSPARSE only at $K = 8$. Its speedups fall to 0.94×, 0.77×, and 0.83× at the other widths. SparseDitto is therefore 1.31×–3.38× faster than DTC-SpMM across the sweep.

4.6 Design space search efficiency

SparseDitto spends 30 minutes per matrix on average with 10 as iteration budget. The limit is LLM inference speed. On GPU, one task runs at most a few dozen compile-and-measure cycles, and each cycle takes seconds to tens of seconds depending on matrix size. The API cost of the full main result is about $20 per matrix.

The search efficiency of SparseDitto is much higher than another automate sparse kernel deisgn space search system, AlphaSparse. Its released code budgets nine hours of search per matrix, for SpMV alone. Under this full budget on `Bai/cdde2`, a matrix with only 4,681 nonzeros, the search compiled and ran 476 candidate kernels, stopped after 50 minutes with its design space exhausted, and its best kernel roughly matches cuSPARSE (1.04×). SparseDitto reaches 4.10× on the same matrix in much less time. The only extra pay is $20. And we believe the efficiency of SparseDitto can be further improve with the acceleration of LLM inference, and the money cost can also be lower with continuous optimization on GPU systems.

The 30-minute is not a floor. Tree-search branches are independent, and the GPU sits idle while the LLM thinks. The branches of one task can therefore run concurrently on a single device, which cuts the wall-clock time toward that of the slowest branch. AlphaSparse's search offers no such slack. Its loop is sequential by construction: each step picks the next candidate from the measured performance

of the previous one, and the GPU must stay free for the measurement itself.

5 Conclusion

The performance of GPU sparse kernels depends jointly on the data representation, the execution schedule, and the hardware mapping, which must all match the sparsity pattern. Our characterization shows that no fixed implementation dominates across patterns, operators, and GPUs. SparseDitto addresses this challenge with a unified design framework that formulates sparse-kernel construction as the joint design of these three components, covering SpMV, SpMM, and SpGEMM. Within this framework, an interpretable energy model ranks established strategies from structural features, an architecture-aware planner explores the design space under target-GPU constraints, and LLM agents automatically generate, verify, and refine each candidate with measurements on the target GPU. SparseDitto outperforms cuSPARSE in evaluated tasks, achieving geometric-mean speedups of 2.68× on an RTX PRO 6000 and 2.79× on an H200. It runs 1.57× to 5.37× faster than four state-of-the-art specialized systems and accelerates full-batch GCN training by up to 3.39×.

References

[1] Rishabh Agarwal, Levi Melnick, Nicholas Frosst, Xuezhou Zhang, Ben Lengerich, Rich Caruana, and Geoffrey E. Hinton. 2021. Neural additive models: Interpretable machine learning with neural nets. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 34. 4699–4711.

[2] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proc. 22nd Int. Conf. on Machine Learning (ICML)*. 89–96.

[3] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proc. 24th Int. Conf. on Machine Learning (ICML)*. 129–136.

[4] Xing Cong, FuKai Sun, YiFan Chen, Chenhao Xie, Yi Liu, and Depei Qian. 2025. CB-SpMV: A Data Aggregating and Balance Algorithm for for Cache-Friendly Block-Based SpMV on GPUs. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 149–160.

[5] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1–1:25.

[6] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. AlphaSparse: generating high performance SpMV codes directly from sparse matrices. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 66, 15 pages.

[7] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 253–267. doi:10.1145/3620666.3651378

[8] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw.* 43, 4, Article 30 (Jan. 2017), 49 pages. doi:10.1145/3017994

[9] Jerome H. Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of Statistics* 29, 5 (2001), 1189–1232.

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

[11] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing* (Goa, India) *(HiPC'07)*. Springer-Verlag, Berlin, Heidelberg, 197–208.

[12] Trevor J. Hastie and Robert J. Tibshirani. 1990. *Generalized Additive Models*. Chapman & Hall/CRC.

[13] Abdullah Al Raqibul Islam, Helen Xu, Dong Dai, and Aydın Buluç. 2025. Improving SpGEMM Performance Through Matrix Reordering and Cluster-wise Computation. *arXiv preprint arXiv:2507.21253* (2025).

[14] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. doi:10.1145/3133901

[15] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. 2014. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. 36, 5 (Jan. 2014), C401–C423. doi:10.1137/130930352

[16] Yann LeCun, Sumit Chopra, Raia Hadsell, Marc'Aurelio Ranzato, and Fu Jie Huang. 2006. A tutorial on energy-based learning. In *Predicting Structured Data*. MIT Press.

[17] Shiyang Li, Zijian Zhang, Winson Chen, Yuebo Luo, Mingyi Hong, and Caiwen Ding. 2026. StitchCUDA: An Automated Multi-Agents

End-to-End GPU Programming Framework with Rubric-based Agentic Reinforcement Learning. In *Proceedings of the 43rd International Conference on Machine Learning (ICML'26)*.

[18] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 339–350. doi:10.1145/2751205.2751209

[19] Duane Merrill and Michael Garland. 2016. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 678–689. doi:10.1109/SC.2016.57

[20] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. 101–110. doi:10.1109/ICPP.2017.19

[21] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: a tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 90–106. doi:10.1145/3503221.3508431

[22] NVIDIA Corporation. 2026. cuSPARSE Library Documentation. https://docs.nvidia.com/cuda/cusparse/

[23] OpenAI. 2026. GPT-5.6: Frontier Intelligence That Scales with Your Ambition. Accessed: 2026-07-31. https://openai.com/index/gpt-5-6/

[24] Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? arXiv:2502.10517 [cs.LG] https://arxiv.org/abs/2502.10517

[25] Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 362–375. doi:10.1145/3332466.3374521

[26] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, Md Amit Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. 2024. MaxK-GNN: Extremely Fast GPU Kernel Design for Accelerating Graph Neural Networks Training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 683–698. doi:10.1145/3620665.3640426

[27] Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proc. 29th ACM Int. Conf. on Supercomputing (ICS)*.

[28] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 65–77. doi:10.1145/3490422.3502357

[29] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 766–780. doi:10.1109/MICRO50266.2020.00068

[30] F. Vázquez, E.M. Garzón, and J.J. Fernández. 2010. A matrix approach to tomographic reconstruction and its implementation on GPUs. *Journal of Structural Biology* 170, 1 (2010), 146–151. doi:10.1016/j.jsb.2010.01.021

[31] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *USENIX Annual Technical Conference (ATC)*.

[32] Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. 2025. Astra: A Multi-Agent System for GPU Kernel Performance Optimization. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.

[33] Min Wu, Huizhang Luo, Fenfang Li, Yiran Zhang, Zhuo Tang, Kenli Li, Jeff Zhang, and Chubo Liu. 2025. HSMU-SpGEMM: Achieving High Shared Memory Utilization for Parallel Sparse General Matrix-Matrix Multiplication on Modern GPUs. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1452–1466.

[34] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. doi:10.1145/3582016.3582047

[35] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proc. 23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*. 94–108.

[36] Xinguo Zhu, Shaohui Peng, Jiaming Guo, Yunji Chen, Qi Guo, Yuanbo Wen, Hang Qin, Ruizhi Chen, Qirui Zhou, Ke Gao, et al. 2025. QiMeng-Kernel: Macro-Thinking Micro-Coding Paradigm for LLM-Based High-Performance GPU Kernel Generation. *arXiv preprint arXiv:2511.20100* (2025).