SABER: SEMANTIC ABSTRACTION BOUNDARIES FOR EFFICIENT REFINEMENT OF SPARSE GPU KERNELS

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU coding agents increasingly optimize kernels through search and execution feedback, but the programming interface presented to the agent remains underexplored. We present SABER, an agent-facing interface for sparse GPU kernels that exposes sparse strategy and coarse hardware intent while delegating concrete GPU realization to a compiler backend. Across 26 workloads spanning six sparse operator groups, SABER records no hard failures versus 11.1% for Direct CUDA, while 92.3% versus 47.4% of trajectories are within 10% of their final search best by proposal 2. Because the reachable implementation spaces differ, we treat this as a full-system comparison. We therefore construct a matched study on ten CSR SpMM/SDDMM workloads in which SABER and a lower-level schedule expose the same 32 executable backend implementations per workload. Across GPT-5.6 Sol, GPT-6 Astra, and MiniMax-M3, SABER reaches shared-reference thresholds earlier and selects implementations closer to the shared finite reference. Configured-backend and private-realization experiments further show that final kernel performance depends on backend quality. These results support treating the agent–compiler programming interface as a first-class systems design choice.

1 INTRODUCTION

AI coding agents increasingly generate and refine GPU kernels using execution feedback. Recent systems study direct kernel generation, iterative refinement, reinforcement learning, and hardware-guided search (Ouyang et al., 2025; Baronio et al., 2025; Dai et al., 2026; Zhang et al., 2025). These efforts primarily improve the model or search procedure. We study a complementary systems question: *what programming interface should a coding agent use when optimizing a GPU kernel?*

Programming interfaces determine which decisions programmers express and which details compilers handle. Coding agents can generate low-level GPU code directly, but doing so requires reasoning about indexing, thread mapping, memory movement, synchronization, reductions, and launch mechanics in addition to the underlying optimization strategy. An interface designed for human programmers or compiler passes therefore need not be the most effective interface for a coding agent.

Sparse GPU kernels make this distinction especially visible. A Direct CUDA interface requires the agent to choose both a sparse strategy and its concrete GPU realization. Structured scheduling interfaces such as CAKE (Ye et al., 2026) remove source-code mechanics but still expose detailed physical execution choices. We instead ask whether the agent can express sparse strategy while a compiler backend handles its concrete GPU realization.

We present SABER, an agent-facing programming interface for sparse GPU kernel optimization. Its design vocabulary captures sparse representation, work organization, ownership, reuse opportunities, reduction, composition, and coarse hardware intent, while operator-specific actions expose the choices relevant to each workload. The backend handles concrete lane mapping, index decoding, storage placement, synchronization, vectorization, instruction selection, and launch configuration. We call this division of responsibility the *abstraction boundary*.

This boundary creates a tradeoff: hiding physical details reduces repeated agent reasoning, but only backend-provided realizations remain reachable. SABER therefore moves rather than removes optimization complexity. Our question is whether this division improves agent search while still allowing the backend to produce competitive kernels.

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.

```
const int start = row_offsets[row];
const int end   = row_offsets[row + 1];
for (int nz = start + warp; nz < end;
     nz += warps_per_block) {
    ...
    const float4 a = s_left[lane];
    const float4 b = r4[lane];
    float sum = a.x*b.x + a.y*b.y
              + a.z*b.z + a.w*b.w;
    for(int offset=16; offset>0; offset>>=1)
        sum += __shfl_down_sync(
            0xffffffffu, sum, offset);
    if (lane == 0) output[nz] = sum;}
```

```
configure_sparse({
  representation : CSR,
  grouping       : SCALAR,
  traversal      : ROW_MAJOR,
  partition      : SAMPLE_ENTRY,
  ownership      : EXCLUSIVE,
  merge          : NONE,
  memory         : DIRECT,
  numeric_core   : SCALAR_FMA,
  tile           : LINEAR
})
// GPU realization delegated to
    backend
```

Figure 1: CSR SDDMM under two agent-facing interfaces. Left: Direct CUDA requires the coding agent to express sparse traversal, GPU cooperation, reduction, and writeback explicitly. Right: SABER exposes sparse strategy and leaves concrete GPU realization to the backend.

This perspective differs from proposing a sparse intermediate representation or lowering mechanism. TACO, SparseTIR, sparse iteration-space transformations, and the Sparse Abstract Machine provide compiler representations and lowering mechanisms for sparse computation (Chou et al., 2018; Senanayake et al., 2020; Ye et al., 2023; Hsu et al., 2023). Our focus is the programming interface between a coding agent and such compiler machinery: which sparse optimization decisions should remain agent-visible, and which implementation decisions should remain below the interface.

We evaluate this question at both system and interface levels. Across 26 workloads covering six sparse operator groups, SABER records no hard failures compared with an 11.1% rate for Direct CUDA, while 92.3% versus 47.4% of trajectories are within 10% of their final search best by proposal 2. Because these systems expose different implementation spaces, we treat this as a full-system comparison. To isolate the agent-facing representation, we additionally construct a matched study in which SABER and a lower-level schedule expose the same 32 executable backend implementations per workload. Candidate cardinality, implementation identity, backend code, workload, numerical contract, and response budget are fixed. Across GPT-5.6 Sol, GPT-6 Astra, and MiniMax-M3, SABER reaches strong candidates earlier and selects implementations closer to the shared finite reference. Separate configured-backend and private-realization experiments show that final kernel performance still depends on backend quality. Our contributions are:

- We introduce SABER, an agent-facing interface that separates sparse optimization strategy expressed by a coding agent from physical GPU realization handled by a compiler backend.
- We develop a matched evaluation methodology that exposes the same executable implementation set through different agent-facing representations, isolating the effect of the interface on search and selection.
- We evaluate the interface across sparse operators and three coding models, and separate agent search behavior from reachable kernel performance through shared finite references, configured backend comparisons, and private realization studies.

2 A Concrete Example: Optimizing CSR SDDMM

We use sampled dense–dense matrix multiplication (SDDMM) as a running example. Given two dense matrices and a sparse set of output coordinates, SDDMM computes a dense-vector dot product at sampled locations. In compressed sparse row (CSR) format, `row_offsets` identify the stored entries in each row and `col_idx` records their column indices. Although the computation is simple to state, a GPU implementation must map sparse work to threads or warps, move operands through the memory hierarchy, combine partial results, synchronize cooperating threads, and construct a valid launch. Figure 1 contrasts how Direct CUDA and SABER expose these responsibilities.

On the Direct CUDA side, the retained model-generated candidate explicitly traverses CSR metadata, assigns sampled entries to warps, performs `float4` loads and a shuffle-based reduction, and selects the writing lane. The coding agent therefore chooses both an optimization strategy and the CUDA mechanics that realize it. The SABER action instead describes the sparse strategy: `CSR` identifies the

representation, ROW_MAJOR the traversal, SAMPLE_ENTRY the logical work unit, and EXCLUSIVE the output ownership, while the remaining fields express coarse memory, numeric, and tiling intent. Concrete lane mapping, vectorization, storage placement, reduction instructions, synchronization, and launch geometry remain below the interface.

This separation allows one agent-visible strategy to admit multiple GPU realizations. The same sampled output can be implemented by a single worker, a warp, a subwarp, or a vectorized subwarp without changing its public sparse semantics. Backend quality therefore remains important: hiding physical choices simplifies the agent-facing problem only when competitive realizations exist below the boundary. Our controlled experiment isolates this interface choice by comparing SABER with a lower-level matched schedule that maps to the same 32 executable backend implementations. In the ten-workload study, SABER responses average 148 output tokens versus 928 for Direct CUDA under GPT-5.6. Separately, in the broader 26-workload system comparison, Direct CUDA produces 20 hard failures among 144 proposals on six CSR SDDMM workloads, while SABER produces none.

3 SABER: An Agent-Facing Programming Interface

3.1 Agent-Facing Abstraction Boundary

The CSR SDDMM example in Section 2 illustrates the central design choice in SABER: the coding agent specifies a sparse computation strategy, while the backend determines its concrete GPU realization. The workload fixes the operator, sparse coordinates, outputs, and numerical contract; the agent controls how sparse work is organized without specifying CUDA lane mapping, storage placement, synchronization, or launch geometry. In the running example, SAMPLE_ENTRY makes one sampled output the logical unit of work, EXCLUSIVE assigns that output to one logical owner, and NONE indicates that no cross-owner merge is required. The backend may nevertheless realize the output with a single lane, a warp, or a subwarp and perform the dot-product reduction internally, as illustrated in Figure 2(a). Reuse opportunities arise from compatible work organization, while register/shared-memory placement remains a backend choice.

We organize the design vocabulary around six recurring dimensions: *representation*; *grouping, partitioning, and traversal*; *ownership and reuse*; *reduction and merge*; *composition*; and *coarse physical intent*. Representation describes how sparse coordinates are encoded, while grouping, partitioning, and traversal define logical work units and their order. Ownership determines responsibility for outputs, and reuse can arise from compatible work organization and operator-specific actions. Reduction and merge describe how partial results combine, composition captures dependencies among sparse operations or intermediate states, and coarse physical intent covers choices such as direct versus staged memory access, numeric-core family, and tile hierarchy. In the evaluated controlled configurations, representation is workload-fixed and the numeric-core field is fixed to scalar-FMA.

These choices specify semantics rather than a schedule. Concrete metadata decoding, thread/lane assignment, vectorization, storage layout, reduction instructions, synchronization, and launch parameters remain backend decisions. A distinction remains public when it changes the logical sparse strategy available to the agent; differences that preserve that strategy while changing only its realization remain below the abstraction boundary. This separation allows a public action to remain stable even when the backend changes how that action is implemented, so improvements in mapping, storage, vectorization, or synchronization need not expand the agent-visible search space. The dimensions therefore describe coherent sparse strategies rather than an arbitrary Cartesian product of independent flags. Table 1 summarizes both sides of this boundary together with the executable scope. Unsupported operator–format paths primarily reflect current backend coverage rather than demonstrated semantic limits of the interface: physical realizations can be incorporated without changing the public action when they preserve the same sparse strategy and numerical contract.

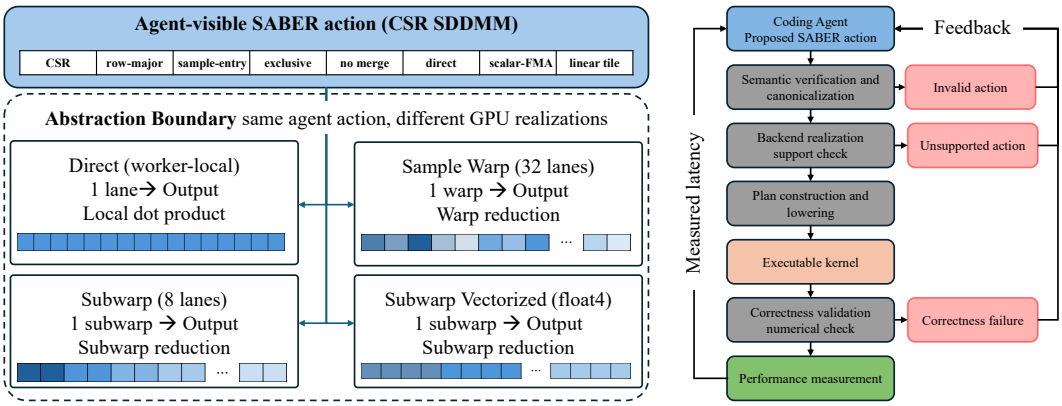
3.2 From Agent Action to GPU Kernel

Figure 2 follows the CSR SDDMM action across the abstraction boundary. Panel (a) shows that one public strategy can be realized by a single worker, a warp, a subwarp, or a vectorized subwarp; these implementations differ in cooperation width, vectorization, and reduction mechanics while preserving the same agent-visible strategy and numerical contract. Panel (b) shows the refinement loop: SABER verifies and canonicalizes the proposed action, checks backend support, lowers it to

Agent-visible choice	Backend realization
Representation	Metadata and address decoding
Work organization	Thread/lane mapping and loop structure
Ownership and reuse	Warp cooperation and storage placement
Reduction and merge	Instructions and synchronization
Composition	Concrete kernel boundaries
Coarse physical intent	Vector widths, resources, and launch parameters

	BSR	CSR	COO
SpMM	✓	✓	✓
SDDMM	✓	✓	✓
SpMV	–	✓	–
Reduction	–	✓	–
SpGEMM	–	✓	–
Attention	–	✓	–

Table 1: SABER's agent-facing boundary and executable scope. Left: agent-visible choices and backend realization. Right: implemented operator–format paths. SpMM and SpMV denote sparse matrix–dense matrix multiplication and sparse matrix–vector multiplication; BSR, CSR, and COO denote block sparse row, compressed sparse row, and coordinate formats, respectively.



(a) One public action, multiple backend realizations.

(b) Validation, lowering, and feedback loop

Figure 2: SABER separates sparse strategy visible to the coding agent from backend GPU realization. (a) One agent-visible action can admit multiple backend realizations. (b) The coding agent proposes an action and receives structured validity, correctness, and performance feedback.

an executable kernel, validates correctness, and measures latency. Structured feedback distinguishes invalid actions, unsupported actions, correctness failures, and measured candidates, allowing the agent to refine its strategy without manipulating CUDA mechanics directly.

3.3 Separating Semantic Validity from Backend Support

SABER separates semantic validity from backend coverage. Semantic validation checks whether the agent-visible choices form a consistent sparse strategy under the fixed workload and numerical contract; canonicalization maps equivalent descriptions to the same public action. A semantically valid action may still be unsupported when the current backend lacks a realization. This distinction lets backend coverage expand or improve without changing the agent-visible action space, provided that the public semantics are preserved.

For example, an action that assigns incompatible ownership and merge semantics is rejected as semantically invalid, whereas a coherent sparse strategy may be unsupported simply because the current backend lacks a corresponding realization. The distinction matters for agent refinement: the former indicates that the proposed strategy should change, while the latter identifies a backend coverage limitation rather than an error in the agent's reasoning.

3.4 Vocabulary Design and Scope

The vocabulary was developed by examining recurring optimization decisions in sparse compiler systems and optimized GPU kernels (Chou et al., 2018; Senanayake et al., 2020; Hsu et al., 2023; Huang et al., 2020; Fan et al., 2024; Xia et al., 2025; Jiang et al., 2024; Ye et al., 2025; Wu et al.,

2025). Across sixty performance-related decisions from six representative systems, recurring choices in representation, work organization, ownership, reuse, reduction, and composition motivated the public vocabulary, while thread/lane mapping, storage placement, vectorization, synchronization, and instruction selection motivated backend responsibilities. Because these systems were used during vocabulary design, this analysis is design evidence rather than an independent coverage result; Section 4.2 therefore evaluates three held-out systems, with decision-level details in Appendix A.

The resulting factorization is intentionally coarser than a physical schedule. We retain a distinction publicly when it changes the logical sparse strategy available to the agent, while recurring choices that only alter how that strategy executes are assigned to backend realization. This criterion, rather than frequency alone, determines where a decision lies relative to the boundary.

The current backend implements the operator–format paths in Table 1; unsupported paths primarily reflect current backend coverage. SABER also does not require a coding model or proposal policy: coding agents, enumeration, heuristics, or other policies can issue actions through the same interface.

4 EVALUATION

Our evaluation separates three questions that can otherwise be confounded: whether the agent-facing vocabulary captures sparse optimization decisions beyond the systems used to design it, whether the representation changes how coding agents search for implementations, and whether the backend can still provide competitive executable kernels. We first evaluate vocabulary transfer on held-out sparse systems, then compare SABER and Direct CUDA across 26 workloads spanning six operator groups as a full-system comparison. To isolate the effect of the agent-facing representation, we additionally compare SABER with a matched lower-level schedule on ten CSR workloads under three coding models, with both representations exposing the same 32 executable backend implementations per workload. Finally, we evaluate independently configured SABER and SparseTIR backends and modify private backend realizations while keeping public SABER actions fixed.

4.1 EXPERIMENTAL SETUP

The primary experiments run on an NVIDIA GeForce RTX 3090 GPU (24 GiB, SM86) with CUDA 13.3. Cross-architecture replay on A100 and H200 is reported in Appendix G; Appendix B gives full software, workload, correctness, search, and timing details.

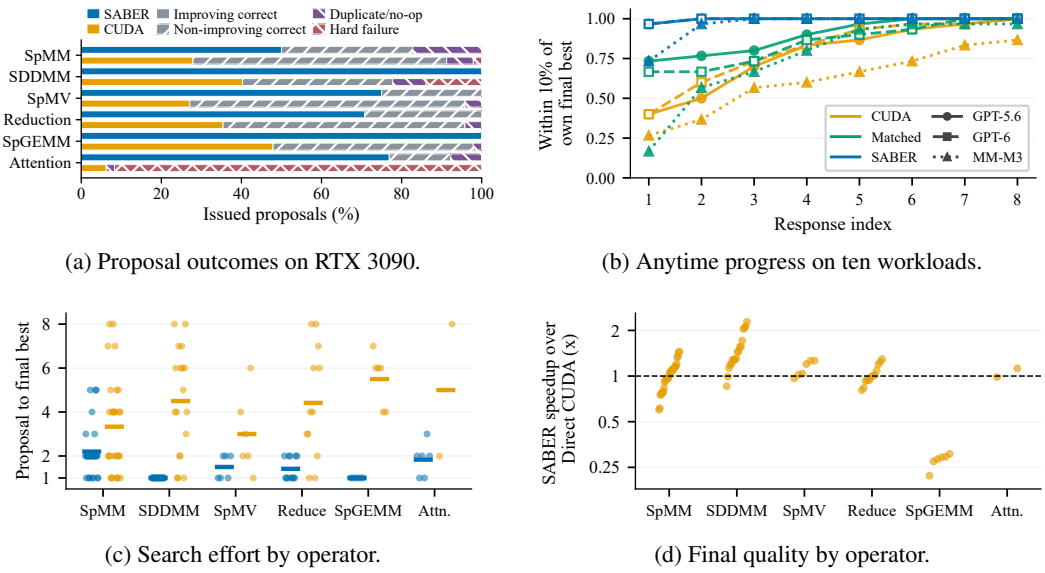
The 26-workload experiment compares Direct CUDA and SABER with three trajectories per workload and interface (78 each). Direct CUDA uses a free-form CUDA source-generation space, whereas SABER uses its structured agent-facing interface. The two conditions therefore need not expose the same reachable implementations, so we treat this as a full-system comparison.

The controlled study uses five CSR SpMM and five CSR SDDMM workloads under GPT-5.6 Sol, GPT-6 Astra, and MiniMax-M3. SABER and the matched schedule expose the same 32 executable candidate identities per workload, drawn from predefined legal configurations, frozen before the runs, and not performance-filtered; candidate i maps to the same backend implementation $K_{w,i}$ under both representations. Backend code, inputs, numerical contracts, response budget, repetitions, and timing protocol are fixed; only the representation presented to the coding model changes. The matched schedule exposes lower-level physical scheduling choices for the same backend identities, rather than SABER's sparse semantic strategy; it is distinct from the configured SparseTIR system in Section 4.5. Each trajectory receives eight response slots; this is a controlled interface study, not a production-scale benchmark.

Throughout the evaluation, *search efficiency* refers to the number of agent responses or proposals required to reach strong candidates and the associated generation burden. We do not claim an end-to-end wall-clock speedup for the complete optimization pipeline.

4.2 HOW DOES THE AGENT-FACING VOCABULARY MAP TO HELD-OUT SPARSE DESIGNS?

After fixing the vocabulary, we classify six source-ordered optimization decisions from each of STile, FlashSparse, and Acc-SpMM, none of which participated in vocabulary design (Fang et al., 2024; Shi et al., 2025; Zhao et al., 2025). Of the eighteen decisions, five map to the public vocabulary,



(a) Proposal outcomes on RTX 3090.

(b) Anytime progress on ten workloads.

(c) Search effort by operator.

(d) Final quality by operator.

Figure 3: Search measurements from two evaluation scopes. (a,c,d) Results from the comparison between SABER and Direct CUDA over 26 workloads. (b) Anytime progress relative to each trajectory's final discovered best in the matched experiment with ten workloads.

eight to backend realization, four fall outside the intended interface, and one is ambiguous: STile's heterogeneous Tensor Core/ELL cover.

The assignments clarify the boundary in practice. Backend decisions include mechanisms such as FlashSparse's ME-BCRS metadata handling and Acc-SpMM's BitTCF encoding, whereas cost-driven optimization policies and relaxed numerical paths fall outside the intended interface. The ambiguous STile case combines heterogeneous Tensor Core and ELL coverage that is only partially captured by existing public concepts. A private classification therefore does not imply a minor implementation detail: some decisions require substantial backend engineering while remaining below the agent-facing boundary. Independent annotations agree on 15/18 decisions ($\kappa = 0.76$).

4.3 How Do the Full Systems Differ in Search Behavior?

Figure 3(a,c,d) summarizes the 26-workload full-system comparison. Both interfaces use the same maximum budget of eight proposals per trajectory, although SABER trajectories may terminate earlier when the finite action domain is exhausted. SABER produces 189 proposals with no hard failures, whereas Direct CUDA produces 624 proposals with 69 hard failures (11.1%). Every SABER trajectory obtains at least one correct candidate, compared with 74/78 Direct CUDA trajectories. The failures are highly operator dependent: Direct CUDA records 5/240 failures for SpMM, 20/144 for SDDMM, and 44/48 for sparse attention, while SABER records none in all six operator groups.

The systems also differ in how quickly useful candidates are found. By proposal 2, 92.3% of SABER trajectories are within 10% of their final search best, compared with 47.4% for Direct CUDA; the mean proposal indices first reaching the final best are 1.628 and 3.986, respectively.

Earlier search progress does not imply faster final kernels. Across 74 valid paired runs, the geometric-mean SABER speedup over Direct CUDA is 0.9915×. SDDMM, SpMV, and reduction favor SABER, whereas SpGEMM strongly favors Direct CUDA at 0.2764×. Because the systems expose different implementation spaces, these results are full-system evidence, not a controlled test of abstraction placement; Appendix C reports raw failure counts and operator-level breakdowns.

4.4 Does Representation Affect Search Across Coding Models?

The matched study isolates representation by exposing the same 32 backend implementations through SABER and a lower-level schedule; Direct CUDA remains a full-system baseline. Table 2 shows

		Search volume		Search behavior			Reliability	Generation burden	
Model	**Interface**	**Resp**	**Succ**	**First**	**At 2**	**Mean best**	**Fmt./Inv./Hard**	**Tokens**	**Gen. s**
GPT-5.6 Sol	CUDA	240	30/30	25/30	15/30	5.60	0/–/11	928	21.3
	Matched	240	30/30	27/30	23/30	3.83	0/9/0	348	10.9
	SABER	240	30/30	29/30	30/30	2.40	0/3/0	148	7.2
GPT-6 Astra	CUDA	240	30/30	27/30	18/30	5.20	0/–/6	948	35.3
	Matched	240	30/30	27/30	20/30	3.67	0/8/0	320	14.5
	SABER	240	30/30	29/30	30/30	2.27	0/2/0	142	9.6
MiniMax-M3	CUDA	240	26/30	20/30	11/30	4.31	34/–/60	2011	17.4
	Matched	240	29/30	20/30	17/30	4.24	12/18/0	620	10.0
	SABER	240	30/30	24/30	29/30	2.83	6/10/0	340	7.8

Table 2: Search results on ten workloads. SABER and the matched schedule share 32 executable candidates per workload; Direct CUDA is a free-form CUDA source-generation baseline. Resp. denotes issued responses; Succ. denotes trajectories with at least one admitted candidate; First denotes first-response admission; At 2 denotes attainment within 10% of the trajectory's final discovered best by response 2; Mean best is the mean response index at which the final discovered best is first reached. Fmt./Inv./Hard denote format, invalid-action, and hard failures; "–" means not applicable.

that SABER reaches its final discovered best at mean response indices 2.40, 2.27, and 2.83 under GPT-5.6, GPT-6, and MiniMax, versus 3.83, 3.67, and 4.24 for the matched schedule. With candidate cardinality, implementation identity, backend code, and timing protocol fixed, these differences reflect search through two representations of the same implementation portfolio.

The representation also affects response reliability and generation burden. Under GPT-5.6 and GPT-6, the matched schedule produces 9 and 8 invalid actions, compared with 3 and 2 for SABER; under MiniMax, the corresponding counts are 18 and 10. Neither structured condition produces a candidate hard failure. SABER responses are also shorter: 148/142/340 mean output tokens versus 348/320/620 for the matched schedule under GPT-5.6, GPT-6, and MiniMax, respectively. Generation time follows the same direction within each model family.

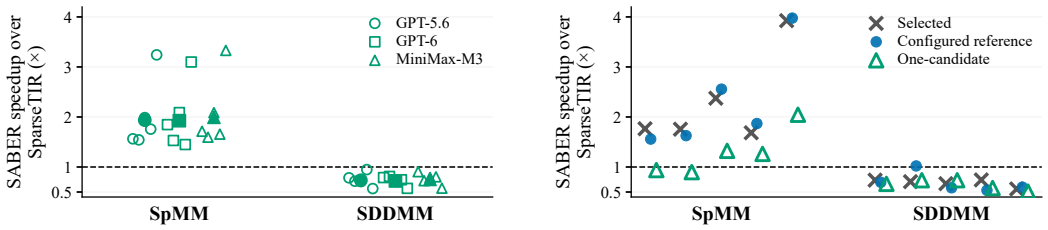
A paired stratified workload bootstrap confirms the search difference: mean reductions in responses to best are 1.40–1.43 across the three coding models, with positive 95% confidence intervals, while response-2 attainment improves by 23.3–40.0 percentage points. The result also holds against the shared finite reference, defined for each workload as the fastest correct implementation in the common 32-candidate portfolio. By response 2, SABER reaches within 10% of that reference in 80.0% of trajectories versus 56.7% for the matched schedule; the corresponding 5% rates are 70.0% and 45.6%. Appendix E reports the complete bootstrap and anytime-reference results.

4.5 How Do Selection and Backend Realization Shape Final Performance?

The preceding experiments concern how the agent searches. We next separate two factors that determine final kernel latency: which implementation is selected within a fixed reachable set, and how competitive the backend realizations in that set are.

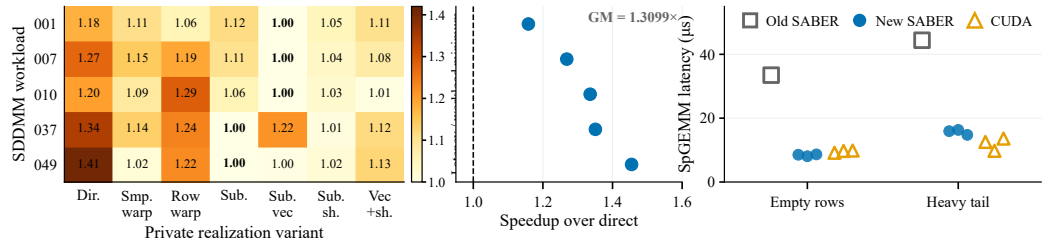
Selection quality under a shared implementation set. For each workload, the fastest correct implementation among the common 32 candidates provides a shared finite reference. Across the six coding-model and operator conditions, SABER obtains selected/reference latency ratios of 1.024–1.063, compared with 1.068–1.137 for the matched schedule. Aggregated over workloads, the SABER speedups over the matched schedule are 1.072, 1.064, and 1.080 under GPT-5.6, GPT-6, and MiniMax, respectively. Because the two controlled representations share candidate cardinality, implementation identity, backend code, and reference optimum, these differences primarily reflect selection within the shared portfolio rather than differences in reachable backend quality.

Configured backend performance. We separately evaluate independently configured SABER and SparseTIR systems, where candidate domains and backend implementations may differ. Figure 4(a) shows that across the three coding models, SABER speedup over SparseTIR is 1.9260–1.9937$\times$ for



(a) Across coding models. (b) Candidate-set controls.

Figure 4: Configured backend performance and candidate-set controls. (a) Selected SABER speedup over SparseTIR across coding model configurations. (b) Selected results from a separate control run, together with configured-reference and one-candidate comparisons. The dashed line denotes parity.



(a) SDDMM private realizations and recovery (b) SpGEMM backend evolution

Figure 5: Backend improvement under unchanged public semantics. (a) SDDMM private realizations and comparison of selected and direct realizations. (b) SpGEMM latency before and after backend replacement. The public SABER action is unchanged in both cases.

SpMM and 0.7216–0.7523× for SDDMM. These measurements characterize configured systems and should not be interpreted as a causal effect of abstraction level.

Figure 4(b) provides candidate-set controls. The configured-reference results are 2.1703× for SpMM and 0.6675× for SDDMM. With one retained candidate per interface, the corresponding speedups are 1.2354× and 0.6383×. Equalizing candidate count therefore does not remove the operator-level direction, although the implementations and backend code are still different. We consequently keep these results separate from the matched interface claim above. Replaying fixed GPT-5.6-selected implementations preserves the same operator-level direction on A100 and H200: SpMM/SDDMM speedups are 1.9588×/0.9323× and 1.9723×/0.8816×, respectively (Appendix G).

Backend realization under unchanged public semantics. The abstraction boundary also allows backend implementations to improve without changing the agent-visible action. Figure 5 changes only private realizations while keeping public SABER semantics fixed. For SDDMM, selecting among private realizations gives a 1.3099× geometric-mean improvement over the original direct realization. For SpGEMM, replacing the private lowering improves SABER by 3.3578× and raises its speedup over Direct CUDA from 0.2764× to 0.9282×.

The heavy-tail SpGEMM workload still favors Direct CUDA, so backend improvement can narrow a performance gap without necessarily eliminating it. Together, these experiments separate two effects of the boundary: the agent-facing representation influences which implementation is selected within a fixed set, while backend realization quality determines how competitive that set can become.

These results highlight two distinct factors in an agent–compiler system. A better agent-facing representation can improve search and selection without repairing a weak backend, while a stronger backend does not by itself ensure that the agent will efficiently identify its best realization. The abstraction boundary therefore affects both what the agent must search over and where implementation quality can be improved independently. This decomposition makes the interface and backend independently improvable: one shapes search within a portfolio, while the other determines the quality of the portfolio itself.

5 Related Work

Agentic GPU optimization and programming interfaces. KernelBench evaluates LLM-generated GPU kernels, while Kevin, CUDA Agent, and CudaForge study iterative optimization with execution or hardware feedback (Ouyang et al., 2025; Baronio et al., 2025; Dai et al., 2026; Zhang et al., 2025). CAKE is most closely related because it also treats the compiler–agent interface as a systems design choice, exposing typed hardware-level scheduling decisions (Ye et al., 2026). Tensor compilers similarly expose structured schedule and tile spaces (Chen et al., 2018; Zheng et al., 2020; Zhu et al., 2022; Shi et al., 2023; Tillet et al., 2019; Lattner et al., 2021). In contrast, SABER places the boundary at sparse strategy and coarse hardware intent, delegating concrete lane mapping, vectorization, storage, synchronization, and launch mechanics to the backend. Its contribution is the agent-facing abstraction boundary and a methodology for evaluating its effect on search and selection.

Sparse compilers and sparse GPU kernels. TACO, sparse iteration-space transformations, Sparse-TIR, and the Sparse Abstract Machine provide representations and lowering mechanisms for sparse computation (Kjolstad et al., 2017; Chou et al., 2018; Senanayake et al., 2020; Ye et al., 2023; Hsu et al., 2023). SABER instead studies which sparse decisions should be exposed to a coding agent and which should remain inside reusable compiler machinery; it is therefore complementary to the underlying sparse IR or lowering mechanism. Specialized GPU kernels optimize formats, load balancing, data movement, and Tensor Core mappings (Huang et al., 2020; Fan et al., 2024; Xia et al., 2025; Fang et al., 2024; Shi et al., 2025; Zhao et al., 2025; Wu et al., 2025), and motivate many of the recurring decisions in SABER's vocabulary.

Efficient attention. Sparse Transformer, Longformer, Reformer, Routing Transformer, BigBird, and MInference exploit structured or dynamic sparsity (Child et al., 2019; Beltagy et al., 2020; Kitaev et al., 2020; Roy et al., 2021; Zaheer et al., 2020; Jiang et al., 2024), while FlashAttention, FlashAttention-2, and FlashInfer emphasize efficient GPU realization (Dao et al., 2022; Dao, 2024; Ye et al., 2025). These systems illustrate the interaction between algorithmic structure, work organization, and GPU realization that motivates the abstraction boundary studied by SABER.

6 Discussion and Conclusion

This work studies programming-interface design for coding agents that optimize sparse GPU kernels. In the matched comparison, SABER and the lower-level schedule share candidate cardinality, implementation identity, backend code, workload, numerical contract, and response budget; only the representation presented to the coding model changes. Under this control, SABER reaches strong candidates earlier and selects implementations closer to the shared finite reference across GPT-5.6, GPT-6, and MiniMax. Direct CUDA remains a full-system baseline rather than part of this control.

SABER shifts rather than eliminates optimization complexity. Sparse work organization, representation, ownership, reduction, composition, reuse, and coarse hardware intent remain agent-visible, while lane mapping, vectorization, storage, synchronization, instruction selection, and launch configuration are delegated to the backend. This suggests a practical boundary criterion: decisions that change the logical sparse strategy should remain public, while changes that preserve that strategy can remain private if the backend provides competitive realizations. The SDDMM and SpGEMM experiments show that such realizations can improve without changing the public action.

The evidence has several limits. The matched study covers 32 candidates per workload over ten CSR workloads and three coding models, and may not generalize to much larger or less structured spaces. The 26-workload experiment spans six operator groups but is a full-system comparison rather than an abstraction-level control. The held-out vocabulary study does not establish vocabulary completeness, and the A100/H200 experiments replay implementations selected on RTX 3090 rather than performing architecture-specific search.

Within these limits, our results support treating the programming interface between a coding agent and compiler backend as a first-class systems design choice. An effective interface should preserve decisions useful for agent search while moving reusable implementation mechanics below the boundary, where they can evolve independently.

AI USE STATEMENT

In this work, we used generative AI tools for research execution, specifically as experimental coding agents to generate kernel candidates. We have not used generative AI tools to prove mathematical claims or generate synthetic datasets; other required-disclosure tasks were not applicable. Additionally, we used generative AI tools for manuscript drafting, editing, and figure scripting. We reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Appendices B– F document workload identities, correctness requirements, search budgets, timing boundaries, aggregation rules, candidate spaces, and configured-backend controls. Cross-architecture replay is reported separately in Appendix G.

REFERENCES

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. Kevin: Multi-Turn RL for generating CUDA kernels. *CoRR*, abs/2507.11948, 2025. URL https://arxiv.org/abs/2507.11948.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, volume 31, 2018.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. URL http://arxiv.org/abs/1904.10509.

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):123:1–123:30, 2018. doi: 10.1145/3276493.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, Wei-Ying Ma, Ya-Qin Zhang, Jingjing Liu, Mingxuan Wang, Xin Liu, and Hao Zhou. CUDA agent: Large-scale agentic RL for high-performance CUDA kernel generation. *CoRR*, abs/2602.24286, 2026. doi: 10.48550/ARXIV.2602.24286. URL https://doi.org/10.48550/arXiv.2602.24286.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with io-awareness. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html.

Ruibo Fan, Wei Wang, and Xiaowen Chu. DTC-SpMM: Bridging the gap in accelerating general sparse matrix multiplication with Tensor Cores. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (eds.), *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pp. 253–267. ACM, 2024. doi: 10.1145/3620666.3651378. URL https://doi.org/10.1145/3620666.3651378.

Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. STile: Searching hybrid sparse formats for sparse deep learning operators automatically. *Proc. ACM Manag. Data*, 2(1):68:1–68:26, 2024. doi: 10.1145/3639323. URL https://doi.org/10.1145/3639323.

Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjolstad. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 710–726. ACM, 2023. doi: 10.1145/3582016.3582051.

Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: general-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In Christine Cuicchi, Irene Qualters, and William T. Kramer (eds.), *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, pp. 72. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00076. URL https://doi.org/10.1109/SC41405.2020.00076.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H. Abdi, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2020.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77:1–77:29, 2017. doi: 10.1145/3133901.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021. doi: 10.1109/CGO51591.2021.9370308.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025. URL https://proceedings.mlr.press/v267/ouyang25a.html.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Trans. Assoc. Comput. Linguistics*, 9:53–68, 2021. doi: 10.1162/TACL_A_00353. URL https://doi.org/10.1162/tacl_a_00353.

Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. A sparse iteration space transformation framework for sparse tensor algebra. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 158:1–158:30, 2020. doi: 10.1145/3428226.

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 (PPoPP)*, pp. 312–325, 2025. doi: 10.1145/3710848.3710858.

Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 701–718. USENIX Association, 2023.

Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pp. 10–19, 2019. doi: 10.1145/3315508.3329973.

Chenpeng Wu, Qiqi Gu, Heng Shi, Jianguo Yao, and Haibing Guan. Samoyeds: Accelerating MoE models with structured sparsity leveraging Sparse Tensor Cores. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 293–310. ACM, 2025. doi: 10.1145/3689031.3717455.

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.

Ran Yan, Youhe Jiang, and Binhang Yuan. Flash sparse attention: An alternative efficient implementation of native sparse attention kernel. *CoRR*, abs/2508.18224, 2025. doi: 10.48550/ARXIV.2508.18224. URL https://doi.org/10.48550/arXiv.2508.18224.

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. SparseTIR: Composable abstractions for sparse compilation in deep learning. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (eds.), *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pp. 660–678. ACM, 2023. doi: 10.1145/3582016.3582047. URL https://doi.org/10.1145/3582016.3582047.

Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. FlashInfer: Efficient and customizable attention engine for LLM inference serving. In *Proceedings of Machine Learning and Systems*, volume 7, 2025.

Zihao Ye, Yingyi Huang, Hongyi Jin, Bohan Hou, Junru Shao, Zhongming Yu, Jinqi Chen, Meghan Cowan, Shiyi Cao, Shanli Xing, Hanfeng Chen, Vinod Grover, Tianqi Chen, and Luis Ceze. CAKE: Compiler-agent co-design for frontier kernel evolution, 2026. URL https://arxiv.org/abs/2608.12629.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Zijian Zhang, Rong Wang, Shiyang Li, Yuebo Luo, Mingyi Hong, and Caiwen Ding. CudaForge: An agent framework with hardware feedback for CUDA kernel optimization. *CoRR*, abs/2511.01884, 2025. doi: 10.48550/ARXIV.2511.01884. URL https://doi.org/10.48550/arXiv.2511.01884.

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, PPoPP 2025, Las Vegas, NV, USA, March 1-5, 2025*, pp. 326–338. ACM, 2025. doi: 10.1145/3710848.3710888. URL https://doi.org/10.1145/3710848.3710888.

Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 863–879. USENIX Association, 2020.

Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. Roller: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 233–248. USENIX Association, 2022.

A Vocabulary Design Details

A.1 Vocabulary Construction

The public vocabulary was developed by examining recurring optimization decisions in sparse GPU implementations. For the detailed decision-level analysis, we use six representative systems: Voltrix-SpMM (Xia et al., 2025), DTC-SpMM (Fan et al., 2024), Flash Sparse Attention (Yan et al., 2025), MInference (Jiang et al., 2024), FlashInfer (Ye et al., 2025), and Samoyeds (Wu et al., 2025). Across these systems, we identified sixty performance-related decisions and recorded whether each decision could be represented directly by one public concept, expressed by composing multiple public concepts, left to backend realization, or placed outside the intended scope of the interface.

Among the sixty decisions, 22 are represented directly by a public concept, 25 are expressed by composing public concepts, four are treated as implementation choices handled by the backend, and nine fall outside the intended scope of SABER. These decisions were used while designing the vocabulary, so the counts describe how the vocabulary was constructed. They should not be interpreted as an independent coverage result or as evidence that the chosen factorization is minimal.

Table 3 summarizes how we distinguish the categories used during vocabulary construction and in the held-out classification.

Table 3: Decision categories used when constructing and evaluating the SABER vocabulary.

Category	**Interpretation**
Public–direct	The decision is represented by one existing public concept.
Public–composed	The decision is represented by combining existing public concepts without introducing a new public concept.
Private	The sparse strategy is unchanged, while the distinction concerns its concrete GPU realization and is therefore handled by the backend.
Outside	The decision changes something intentionally not controlled by the public interface, such as the numerical contract or an external optimization policy.
Ambiguous	The fixed vocabulary does not provide enough information to establish whether the decision should be represented publicly or handled below the boundary.

The CSR SDDMM example used in the main text illustrates the resulting public interface. Its action is `configure_sparse_execution`, with CSR representation, scalar grouping, row-major traversal, sample-entry partitioning, exclusive ownership, no merge, direct memory, scalar-FMA execution, and a linear tile hierarchy. These fields describe the sparse strategy and coarse implementation intent. Cooperative width, metadata decoding, vectorized loads, storage placement, and other physical details are selected by the backend.

The vocabulary changed during development when existing concepts were insufficient. In particular, SpGEMM required a relational-composition refinement. The version used for the held-out classification below was fixed after this change; none of the three held-out systems was used to modify that version.

A.2 Held-Out Classification

STile, FlashSparse, and Acc-SpMM were not used when constructing the vocabulary. For each system, we retain six optimization decisions in source order and classify each decision according to the fixed SABER boundary. We first identify what the source implementation changes and then determine whether that change is represented by the public vocabulary, belongs to backend realization, falls outside the intended interface, or cannot be assigned unambiguously.

The eighteen held-out decisions are independently classified by two analysts using the fixed category definitions in Table 3. The second analyst does not observe the first analyst's labels before classification. The two label sets agree on 15/18 decisions (83.3%), with Cohen's $\kappa = 0.76$. Disagreements are reviewed after the independent pass; Table 4 reports the adjudicated labels.

Five of the eighteen decisions map to the public vocabulary, eight are assigned to backend realization, four fall outside the intended public interface, and one remains ambiguous. The ambiguous case

Table 4: Held-out sparse optimization decisions and their position relative to the fixed SABER boundary. Public includes both direct and composed representations. Private denotes backend realization; Outside denotes decisions beyond the intended public interface.

Source	Source-ordered decision	Assignment
STile	Reorder coordinates with recovery maps	Public
	Cover an operation with Tensor Core and ELL tiles	Ambiguous
	Select tiles by estimated cost per useful work	Outside
	Replace earlier tiles with a whole-row choice	Outside
	Atomic completion for overlapping tile outputs	Private
	Shared-memory padding for bank conflicts	Private
FlashSparse	Group nonzeros into eight-row vectors	Public
	Swap operands and transpose register interpretation	Private
	Remap lanes for coalesced loads and stores	Private
	Split SDDMM output blocks for subsequent SpMM	Public
	ME-BCRS metadata and neutral-tail decoding	Private
	Use a relaxed TF32 numerical path	Outside
Acc-SpMM	Data-affinity reordering with coordinate recovery	Public
	Encode tile positions in a BitTCF bitmap	Private
	Place metadata in shared memory and registers	Private
	Double-buffer metadata loads and MMA	Private
	Split and combine row-window work	Public
	Choose balancing with an imbalance/cost policy	Outside

is STile's heterogeneous Tensor Core/ELL cover. Existing ownership and composition concepts describe parts of this design, but the fixed vocabulary does not establish a general representation for an arbitrary heterogeneous cover.

Several Private assignments also require substantial implementation work. For example, FlashSparse's ME-BCRS decoding and Acc-SpMM's BitTCF metadata handling are classified below the public boundary, but neither is implemented as a port in SABER. The classification therefore describes where these decisions would belong under the boundary.

For reproducibility, the source versions used for this classification are STile commit `604b8e8675fdf3f7350a68fc13ed2810d01a56cc`, FlashSparse (Shi et al., 2025), and Acc-SpMM (Zhao et al., 2025).

B Experimental Setup and Workload Definitions

B.1 Hardware, Software, and Workloads

All primary measurements are run on an NVIDIA GeForce RTX 3090 GPU with 24 GiB memory and SM86 / Compute Capability 8.6. The machine uses Driver 616.56, CUDA 13.3, and nvcc 13.3.73.

The broad experiment with 26 workloads contains six BSR SpMM, four CSR SpMM, six CSR SDDMM, two CSR SpMV, four SUM/MAX reduction, two CSR SpGEMM, and two sparse attention workloads. Each interface is run three times on every workload, giving 78 trajectories per interface.

The controlled experiment uses the ten CSR workloads listed in Table 5: five SpMM and five SDDMM workloads. The same frozen FP32 inputs are shared across all conditions. These workloads were chosen because the compared interfaces support the same CSR operators and numerical contract; they are not a random sample of sparse workloads.

B.2 Coding Model Configurations

The controlled experiment with ten workloads uses GPT-5.6 Sol and GPT-6 Astra with medium reasoning and MiniMax-M3 with Think-Off / none. The recorded model identifiers are `gpt-5.6-sol`, `gpt-6-astra`, and `MiniMax-M3`; MiniMax client metadata records `none` for reasoning. The

Table 5: Exact CSR workloads shared by all conditions. Dimensions use each operator's original convention. Inputs are FP32; all ten use absolute and relative tolerance 10^{-4}.

Workload	M	N	K	nnz	Seed
spmm-csr-001	96	32	128	192	8201
spmm-csr-010	96	32	128	3072	8210
spmm-csr-007	96	32	128	1536	8207
spmm-csr-037	192	32	192	576	8237
spmm-csr-049	256	64	256	1024	8249
sddmm-csr-001	96	128	32	192	9201
sddmm-csr-010	96	128	32	3072	9210
sddmm-csr-007	96	128	32	1536	9207
sddmm-csr-037	192	192	32	576	9237
sddmm-csr-049	256	256	64	1024	9249

service does not expose an independent server-side identifier or deterministic generation seed, so we report the client-visible configuration.

Coding model configuration for the broad experiment. The experiment with 26 workloads uses GPT-5.6 Sol with Medium reasoning under the archived codex-cli 0.151.0-alpha.7.2 configuration. Each workload condition is repeated three times with a maximum of eight agent proposals per trajectory. The same coding model configuration and proposal budget are used for SABER and Direct CUDA.

Each trajectory starts from a fresh process and contains only its own previous responses and feedback. The model receives no result from another trajectory, no previously discovered winning implementation, no repository tools, and no persistent memory. Workload input, numerical contract, response budget, and feedback structure are fixed within each comparison.

B.3 Interfaces Presented to the Coding Agent

Table 6 summarizes the three conditions used in the controlled experiment with ten workloads. Direct CUDA is a free-form CUDA source-generation baseline. The matched schedule and SABER conditions form the abstraction control: both address the same executable implementation portfolio but expose it through different representations presented to the coding agent.

Agent-visible contract. The agent-visible schema is fixed before all controlled runs. For each operator, the schema specifies the legal fields, their allowed values, applicability constraints, and the structured feedback returned after validation. The matched-schedule condition likewise uses a fixed physical scheduling schema. The coding models do not receive backend-private choices such as concrete lane mapping, storage placement, synchronization, or launch configuration under the SABER condition.

In Direct CUDA, the coding agent writes CUDA source and is responsible for both the optimization strategy and the CUDA mechanics that realize it. The matched schedule condition is a controlled representation at a lower level, not a claim about the full SparseTIR compiler stack. It exposes physical choices corresponding to the shared implementation portfolio described below. Once a legal action is parsed, it maps to one canonical implementation identity $K_{w,i}$.

In SABER, the coding agent proposes a structured semantic action specifying operator-specific choices in representation, grouping, traversal, partitioning, ownership, reduction or merge behavior, composition, and coarse hardware intent. Reuse opportunities may be captured indirectly through these choices rather than through a standalone action field. The verifier checks legality and canonicalization removes equivalent syntactic forms before the action maps to the same canonical implementation portfolio.

The independently configured SparseTIR system used for the configured backend performance study is described separately in Appendix F; it is not the matched schedule condition used for the matched abstraction control.

Table 6: Conditions presented to the coding agent in the controlled experiment with ten workloads.

	Direct CUDA	**Matched schedule**	**SABER**
Agent output	CUDA source code	Physical scheduling action	Structured sparse semantic action
Sparse indexing	Implemented in generated source	Exposed through physical scheduling fields	Specified semantically; concrete decoding is private to the backend
Thread/lane mapping	Chosen directly in source	Visible to the coding agent	Private to the backend
Memory realization	Explicit loads, stores, and buffers	Physical choices visible to the coding agent	Coarse memory intent is public; concrete realization is private to the backend
Reduction realization	Implemented directly	Physical choices visible to the coding agent	Ownership and merge are public; instructions and synchronization are private
Candidate space	Free-form CUDA source generation	32 shared identities per workload	32 shared identities per workload
Execution result	Compile, correctness check, and latency	Map to shared identity, correctness check, and latency	Verify, map to shared identity, correctness check, and latency

Table 7: Compact summary of the agent-visible SABER action schema across the evaluated configurations. The controlled CSR experiment uses the applicable subset; field applicability is operator- and representation-dependent.

Field	**Agent-visible choice**	**Backend-private realization**
representation	`bsr`, `csr`, `coo` (workload-fixed in evaluated runs)	Metadata decoding and address calculation
grouping / traversal	`block`/`scalar`; `block_row_major`/`row_major`/ `coordinate_major`	Thread/lane mapping and loop structure
partition / ownership	`complete_rows`, `row_chunks`, `sparse_entries`, `sample_entries`; `exclusive`/`partial`	CTA/warp/subwarp assignment and writeback mapping
reuse	Expressed through compatible work organization and operator-specific actions	Register/shared-memory placement and buffering
reduction / merge	`none`, `add`, `associative`	Shuffle/shared-memory/atomic realization and synchronization
composition	Operator-specific stage/support choices for attention and SpGEMM	Kernel boundaries, fusion, and intermediate materialization
memory	`direct`, `stage_sparse`	Concrete loads/stores, shared-memory layout, and barriers
numeric core	`scalar_fma` in the evaluated configurations	Instruction selection and vectorization
tile	`block_row_x_n_tile`, `block_chunk_x_n_tile`, `linear`	Concrete tile sizes and launch geometry

B.4 Matched Candidate Spaces

For each workload w, the controlled comparison uses a frozen portfolio of 32 executable backend implementations,

$$\mathcal{K}_w = \{K_{w,1}, \ldots, K_{w,32}\}.$$

Both SABER and the matched-schedule condition are restricted to this same finite portfolio. After parsing and canonicalization, an accepted action under either representation resolves to one member of $\mathcal{K}_w$; once the backend identity is fixed, both conditions execute the corresponding backend implementation.

Table 8: Shared candidate domains in the controlled interface comparison.

Operator	Matched schedule	SABER
SpMM	32	32
SDDMM	32	32

The two conditions therefore differ in the representation presented to the coding agent rather than in the executable portfolio. SABER exposes sparse strategy and coarse hardware intent, whereas the matched schedule exposes physical scheduling choices at a lower level. Candidate cardinality, reachable backend identities, backend code, workload input, numerical contract, response budget, and timing protocol are held fixed across the controlled comparison. Variation among multiple private realizations of a public SABER action is evaluated separately in Section 4.5 and is not varied within this experiment.

Each trajectory receives eight response slots. Because the shared domain contains 32 candidates, exhaustion of the candidate space does not occur within the response budget.

Portfolio construction. The portfolio is defined at the backend-implementation level before the controlled runs. Its members are drawn from predefined legal backend configurations rather than selected according to measured latency, and no performance-based filtering is applied when forming the shared portfolio. Candidate membership is then frozen and is not changed in response to model outputs or performance observed during the experiment.

Each backend identity is exposed through both interfaces: SABER describes the corresponding sparse strategy and coarse hardware intent, while the matched schedule describes the associated physical scheduling choices at a lower level. Canonicalization precedes backend-identity lookup so that syntactic variants do not create additional executable candidates. The remainder of Appendix B specifies the workload definitions, coding-model configurations, response procedure, correctness criteria, and timing protocol used in the controlled comparison.

B.5 Response Procedure and Feedback

A controlled trajectory contains at most eight responses. Each response occupies one response slot. A response that yields one unambiguous, in-domain candidate is accepted for evaluation; duplicate candidate identities remain part of the response sequence. A response from which no candidate can be extracted is recorded as a format failure, while an extractable action that does not map to the declared candidate domain is recorded as an invalid interface action for this controlled experiment. This category denotes failure to map into the fixed experimental domain and should not be interpreted as a general claim of semantic invalidity.

A candidate is admitted only after successful build, execution, and numerical validation. An admitted candidate that does not improve the current best latency is counted as correct but non-improving. Canonically equivalent candidate identities count as duplicates. A candidate hard failure occurs when an in-domain candidate fails build, execution, or numerical validation and therefore receives no admitted latency.

For MiniMax, the format/invalid/hard failure counts are 12/18/0 for the matched schedule and 6/10/0 for SABER. For GPT-5.6, the corresponding counts are 0/9/0 and 0/3/0; for GPT-6 they are 0/8/0 and 0/2/0. Direct CUDA retains its separately measured source generation failure counts reported in Appendix D.

B.6 Correctness

All controlled runs over ten workloads use identical frozen FP32 inputs and the same CPU reference outputs. A candidate is admitted only if its output has the expected shape, contains only finite values, and satisfies

$$\mathrm{atol} = \mathrm{rtol} = 10^{-4}.$$

A numerical or execution failure receives no performance latency.

Build, correctness validation, and accepted performance measurement occur in that order. Native input serialization preserves the same FP32 values used by the reference implementation, and outputs are checked again after final measurement.

The same validation policy for finite outputs is used throughout the main experiments. A candidate is admitted only after passing its declared numerical contract, and no failed candidate contributes a performance latency.

B.7 TIMING

Selected candidates are remeasured after search under a common preprocessed timing boundary for a complete device operation. The measured interval contains every GPU kernel and required output reset for one complete invocation. It excludes model generation, parsing, compilation, memory allocation, host–device transfers, and reusable representation conversion.

For the matched schedule and SABER conditions, a selected canonical identity $K_{w,i}$ is replayed through the same backend implementation path. The representation presented to the coding agent is therefore no longer present in the timed kernel once the identity is fixed. Direct CUDA uses its generated native CUDA executable and is retained only as a full system baseline. The independently configured SparseTIR measurements use their original TVM execution path and are reported separately in Appendix F.

Each timing batch begins with five explicit warmups. We then record twenty CUDA event samples, where each sample measures twenty complete invocations and is divided by twenty. The batch value is the median of these twenty sample means.

Each selected trajectory is measured in three batches. Its reported latency is the median of the three batch medians. The value for each workload is then the median of the three trajectory latencies, and a workload/interface/model cell is reported only when all three trajectory measurements are valid.

Across the controlled experiment, 265 trajectories produce a selected candidate and therefore yield 795 valid replay batches. These correspond to 86/90 complete workload/interface/model cells. Direct CUDA under MiniMax has seven complete workload cells; the matched schedule and SABER conditions provide 29/30 and 30/30 complete workload cells across the three model configurations, respectively.

CUDA event intervals may include short gaps caused by host launch issuance. They should therefore be interpreted as complete device operation latency, not kernel execution time at the instruction level. During search, each admitted candidate is timed independently within its trajectory, and the measured latency from that evaluation is returned as feedback to the coding agent; the same candidate identity may therefore exhibit small timing variation across trajectories. Latency measured during search is never substituted for final replay measurement.

B.8 AGGREGATION AND MISSING RESULTS

For n paired workloads, speedups are aggregated with the geometric mean

$$\exp\left(\frac{1}{n}\sum_{w=1}^{n}\log\frac{L_{B,w}}{L_{A,w}}\right),$$

where A is the method named before "speedup over" and B is the comparison method. A value above one therefore means that the method named first is faster. For these controlled workload-level aggregates, each workload receives equal weight. For complete conditions, results by operator use five workload pairs and the overall result uses ten. MiniMax Direct CUDA has five complete SpMM workloads, two complete SDDMM workloads, and seven complete workloads overall. Missing cells are not imputed and all affected ratios are reported with their pair counts.

The three repeated trajectories characterize variation across agent search runs. Repeated timing samples characterize measurement variation. Neither is treated as an additional independent workload.

Attainment by response 2 asks whether the best correct candidate seen by response two is within 10% of the best candidate seen during search for that trajectory. A prefix with no correct candidate does not attain the threshold.

Response to best is the first response index at which a trajectory reaches its best latency seen during search. This quantity is reported only for trajectories that eventually produce a correct candidate.

When selected candidates and shared reference candidates are timed in different phases, candidate identity is checked separately from latency. A timing ratio different from one for the same identity is treated as timing variation across phases rather than missed selection.

C Experiment with 26 Workloads

C.1 Workload Definitions

Table 9 lists the frozen workloads used in the broad experiment with 26 workloads, including operator, format, shape, sparsity, and seed.

Table 9: Workloads in the experiment with 26 workloads. Shape denotes $M \times K \times N$ for SpMM/SpGEMM, $M \times N \times K$ for SDDMM, $M \times N$ for SpMV/reduction, and (L, d_{qk}, d_v) for attention. The nnz column counts stored scalar entries (sampled positions for SDDMM/attention); SpGEMM lists $(\mathrm{nnz}_A, \mathrm{nnz}_B)$. Seed pairs denote (A, B) for SpGEMM and (support, dense inputs) for attention. IDs are presentation aliases for the frozen workload identifiers.

ID	Operator	Format	Shape	nnz	Seed
bsr-spmm-001	SpMM	BSR (4×4)	$256 \times 256 \times 64$	1024	8048
bsr-spmm-002	SpMM	BSR (4×4)	$256 \times 256 \times 64$	1024	8049
bsr-spmm-003	SpMM	BSR (4×4)	$256 \times 256 \times 64$	8192	8054
bsr-spmm-004	SpMM	BSR (4×4)	$256 \times 256 \times 64$	8192	8055
bsr-spmm-005	SpMM	BSR (16×16)	$256 \times 256 \times 64$	16384	8058
bsr-spmm-006	SpMM	BSR (16×16)	$256 \times 256 \times 64$	16384	8059
csr-spmm-001	SpMM	CSR	$256 \times 256 \times 64$	1024	8248
csr-spmm-002	SpMM	CSR	$256 \times 256 \times 64$	1024	8249
csr-spmm-003	SpMM	CSR	$256 \times 256 \times 64$	16384	8258
csr-spmm-004	SpMM	CSR	$256 \times 256 \times 64$	16384	8259
csr-sddmm-001	SDDMM	CSR	$2048 \times 2048 \times 128$	524288	11012
csr-sddmm-002	SDDMM	CSR	$2048 \times 2048 \times 32$	65536	11013
csr-sddmm-003	SDDMM	CSR	$2048 \times 2048 \times 64$	262144	11014
csr-sddmm-004	SDDMM	CSR	$2048 \times 2048 \times 128$	524288	11015
csr-sddmm-005	SDDMM	CSR	$2048 \times 2048 \times 32$	65536	11016
csr-sddmm-006	SDDMM	CSR	$2048 \times 2048 \times 64$	262144	11017
csr-spmv-001	SpMV	CSR	512×512	2190	12002
csr-spmv-002	SpMV	CSR	1024×1024	10240	12003
reduce-sum-001	Reduction (sum)	CSR	512×512	2190	12002
reduce-sum-002	Reduction (sum)	CSR	1024×1024	10240	12003
reduce-max-001	Reduction (max)	CSR	512×512	2190	12002
reduce-max-002	Reduction (max)	CSR	1024×1024	10240	12003
spgemm-001	SpGEMM	CSR	$64 \times 64 \times 64$	$(108, 128)$	$(13001, 14001)$
spgemm-002	SpGEMM	CSR	$96 \times 96 \times 96$	$(188, 192)$	$(13002, 14002)$
attention-001	Attention	CSR	$(128, 32, 32)$	546	$(15002, 16002)$
attention-002	Attention	CSR	$(80, 16, 24)$	272	$(15003, 16003)$

C.2 Search Results

The experiment with 26 workloads uses the workload definitions in Table 9 and the coding model configuration specified in Appendix B.2. It contains 78 trajectories for SABER and 78 for Direct CUDA, with three runs per workload and interface. Across these runs, SABER produces 189 proposals and Direct CUDA produces 624.

Among the SABER proposals, 168 are correct nonduplicates: 118 improve the current best latency and 50 are correct but not improving. The remaining 21 proposals are canonical duplicates. No candidate hard failure is recorded for SABER.

Table 10: Proposal outcomes in the experiment with 26 workloads on RTX 3090. Counts are over issued proposals.

Outcome	SABER	Direct CUDA
Improving correct	118	198
Non-improving correct	50	321
Duplicate / no-op	21	36
Numerical failure	0	54
Source / parse failure	0	11
Compile failure	0	3
Runtime / timeout	0	1
Total	189	624

Table 11: Results by operator for the experiment with 26 workloads. Mean proposals to best condition on successful trajectories. The performance column reports SABER speedup over Direct CUDA; values above one favor SABER and values below one favor Direct CUDA.

	Mean proposals to best		CUDA hard	SABER speedup	Valid
Operator	SABER	Direct CUDA	failures	GM	pairs
SpMM	2.20	3.33	5/240	0.9883	30
SDDMM	1.00	4.50	20/144	1.4367	18
SpMV	1.50	3.00	0/48	1.1179	6
Reduction	1.42	4.42	0/96	1.0121	12
SpGEMM	1.00	5.50	0/48	0.2764	6
Attention	1.83	5.00	44/48	1.0515	2

Direct CUDA produces 519 correct nonduplicate candidates, including 198 improving and 321 correct candidates without improvement. The remaining proposals include 36 duplicates or no-ops and 69 candidate hard failures. These failures consist of 54 numerical failures, eleven source or parsing failures, three compile failures, and one runtime or timeout failure.

Table 10 summarizes these proposal outcomes. In this broad experiment, "hard failure" is an umbrella term for a proposal that fails to yield an admitted latency, including source/parse, compile, runtime, and numerical failures; the controlled experiment uses the more granular taxonomy defined in Appendix B.5.

All 78 SABER trajectories obtain at least one correct candidate. Direct CUDA does so in 74 of 78 trajectories. The four unsuccessful Direct CUDA trajectories occur in sparse attention and remain part of the reliability denominator; they are excluded only from comparisons that require a finite candidate latency.

By proposal two, 72 of 78 SABER trajectories are within 10% of their own eventual best candidate seen during search, compared with 37 of 78 Direct CUDA trajectories. These correspond to attainment rates at proposal 2 of 92.3% and 47.4%, respectively.

Attainment at proposal 2 is different from the first proposal that exactly reaches the final search best. The former accepts any candidate within 10% of the latency of the final best candidate, while the latter records the first exact occurrence of that best candidate during search. Exhaustion of the finite domain is treated as trajectory termination and is not padded with additional proposals.

C.3 Results by Operator

Table 11 reports search effort and final latency by operator. Mean proposals to best are conditioned on trajectories that obtain a correct candidate. Speedups use only workload/repetition pairs for which both interfaces have valid latency measurements.

The failure rate varies substantially by operator. Direct CUDA records 5 failures among 240 SpMM proposals, 20 among 144 SDDMM proposals, and 44 among 48 sparse attention proposals. No

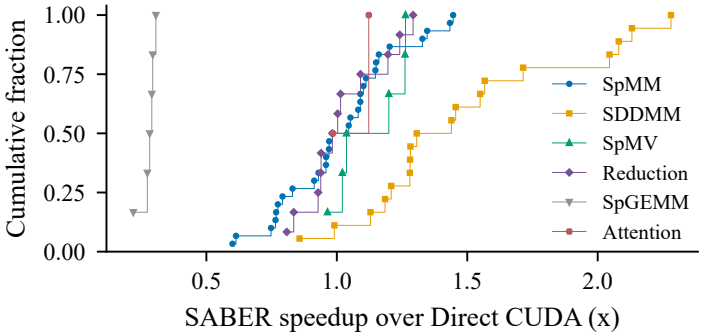


Figure 6: Empirical distribution of final SABER speedup over Direct CUDA in the experiment with 26 workloads. The distribution contains 74 valid paired measurements from individual runs across 25 workloads; repeated runs of the same workload are not independent workloads. Values above one favor SABER, while values below one favor Direct CUDA.

candidate hard failures occur in SpMV, reduction, or SpGEMM. SABER records none in any of the six operator groups.

Search effort and final latency do not always move in the same direction. SDDMM, SpMV, and reduction have SABER speedup over Direct CUDA above one, while SpGEMM strongly favors Direct CUDA despite SABER reaching its selected candidate with fewer proposals. The SABER speedup over Direct CUDA is 0.2764 over six valid workload/repetition pairs.

The geometric mean SABER speedup over Direct CUDA is 0.9883 for SpMM over thirty pairs. The corresponding BSR and CSR speedups are 0.9546 and 1.0411, respectively. Across all successful paired measurements, the overall SABER speedup over Direct CUDA is 0.9915. This value is computed directly from all valid paired measurements rather than from the rounded geometric means by operator.

Sparse attention has only two valid paired performance measurements because four Direct CUDA trajectories do not obtain a correct candidate. Its reported SABER speedup over Direct CUDA therefore describes those successful pairs only and should not be read as a complete comparison for the operator.

D Experiment with Ten Workloads Across Agent-Facing Interfaces

D.1 Search Behavior

The experiment with ten workloads contains three coding model configurations, three interfaces, ten workloads, and three trajectories per condition, for a total of 270 trajectories. Direct CUDA remains a free-form CUDA source-generation baseline. The other two conditions form the matched comparison: SABER and the matched schedule representation expose the same 32 candidate identities per workload, with candidate i mapping to the same executable backend implementation under both representations.

Each trajectory receives a common budget of eight responses. The controlled conditions therefore contain 720 responses each across the three coding model configurations, while the Direct CUDA baseline contains 720 responses.

Table 12 summarizes search behavior.

Across the experiment with ten workloads, correct candidates are obtained in 265 of the 270 trajectories. The four unsuccessful Direct CUDA trajectories remain under MiniMax, while the matched schedule contributes one additional unsuccessful trajectory and SABER succeeds in all 90 controlled trajectories.

Table 12: Search results for the experiment with ten workloads. The SABER and matched schedule rows form the controlled comparison; Direct CUDA is a free-form CUDA source-generation baseline.

		Search volume		Search behavior			Reliability	Generation burden	
Model	**Interface**	**Resp.**	**Succ.**	**First**	**At 2**	**Mean best**	**Fmt./Inv./Hard**	**Tokens**	**Gen. s**
GPT-5.6 Sol	CUDA	240	30/30	25/30	15/30	5.60	0/–/11	928	21.3
	Matched	240	30/30	27/30	23/30	3.83	0/9/0	348	10.9
	SABER	240	30/30	29/30	30/30	2.40	0/3/0	148	7.2
GPT-6 Astra	CUDA	240	30/30	27/30	18/30	5.20	0/–/6	948	35.3
	Matched	240	30/30	27/30	20/30	3.67	0/8/0	320	14.5
	SABER	240	30/30	29/30	30/30	2.27	0/2/0	142	9.6
MiniMax-M3	CUDA	240	26/30	20/30	11/30	4.31	34/–/60	2011	17.4
	Matched	240	29/30	20/30	17/30	4.24	12/18/0	620	10.0
	SABER	240	30/30	24/30	29/30	2.83	6/10/0	340	7.8

For GPT-5.6, the mean response index reaching the final discovered best is 5.60 for Direct CUDA, 3.83 for the matched schedule representation, and 2.40 for SABER. The corresponding values for GPT-6 are 5.20, 3.67, and 2.27. For MiniMax, they are 4.31, 4.24, and 2.83. These means condition on trajectories that eventually produce a correct candidate.

Attainment by response 2 shows the same direction in the controlled comparison. For GPT-5.6, 23/30 matched-schedule trajectories and 30/30 SABER trajectories are within 10% of their final discovered best by response two. The corresponding counts are 20/30 and 30/30 for GPT-6, and 17/30 and 29/30 for MiniMax.

Because the two controlled conditions expose the same candidate cardinality and the same reachable implementation identities, these differences in search effort cannot be attributed to candidate set size or backend coverage. Direct CUDA remains a free-form CUDA source-generation condition and is therefore reported as a separate full system baseline.

D.2 Response Parsing and Candidate Failures

Response format failures are recorded separately from candidate failures. A format failure indicates that no unambiguous candidate can be extracted from the model response. A candidate hard failure occurs only after a candidate has been extracted and then fails during build, execution, or numerical validation.

For GPT-5.6, the matched schedule and SABER conditions produce 0/9/0 and 0/3/0 format/invalid/hard failures, respectively. For GPT-6, the corresponding counts are 0/8/0 and 0/2/0. Thus, the GPT configurations produce no format or candidate hard failures, but the representation at a lower level produces more actions outside the declared domain.

MiniMax exhibits additional response format variation. Direct CUDA retains 34 format failures and 60 candidate hard failures. In the matched conditions, the format/invalid/hard failure counts are 12/18/0 for the matched schedule and 6/10/0 for SABER. Because all candidate identities within the declared domain belong to the prevalidated shared portfolio, the matched conditions have no candidate hard failures.

The response reader accepts an exact JSON object, a single fenced JSON object, or one unambiguous top-level JSON object. It does not rewrite candidate semantics. In the matched conditions, an extracted candidate must also map to one of the declared 32 candidate identities; malformed responses or responses outside the declared domain are not silently repaired.

D.3 First Response Results

Table 13 reports whether the first response yields an admitted candidate and, conditional on admission, the latency of that candidate measured during search.

Table 13: First response diagnostics. "Admitted" means that the first response yields a valid correct candidate; latency is the geometric mean over admitted first-response candidate latencies.

Model	Interface	Admitted	Not admitted	Latency (μs)
GPT-5.6	Direct CUDA	25/30	5/30	8.238
	Matched schedule	27/30	3/30	9.66
	SABER	29/30	1/30	8.91
GPT-6	Direct CUDA	27/30	3/30	8.426
	Matched schedule	27/30	3/30	9.74
	SABER	29/30	1/30	8.88
MiniMax-M3	Direct CUDA	20/30	10/30	9.208
	Matched schedule	20/30	10/30	9.91
	SABER	24/30	6/30	9.04

Table 14: Selection quality relative to the shared finite reference over the 32-candidate portfolio. For each model, operator, and interface, Sel./ref. is the geometric mean of $L_{\mathrm{selected},r}/L_{\mathrm{ref},w(r)}$ over successful selected trajectories r for that interface. Identity-match counts use the same successful-trajectory set. These interface-specific aggregates are not paired across interfaces.

		SpMM		SDDMM	
Model	Interface	Sel./ref.	ID match	Sel./ref.	ID match
GPT-5.6	Matched schedule	1.082	10/15	1.137	7/15
	SABER	1.028	13/15	1.041	12/15
GPT-6	Matched schedule	1.071	11/15	1.122	8/15
	SABER	1.024	14/15	1.036	13/15
MiniMax-M3	Matched schedule	1.115	8/15	1.068	6/14
	SABER	1.047	12/15	1.063	11/15

The first response comparison equalizes the number of agent attempts observed. Within the matched comparison between SABER and the schedule, it also holds candidate cardinality and reachable implementation identities fixed. It is therefore a direct diagnostic of how often each representation yields a useful first optimization decision.

The latency column answers a different question. It conditions on the first response yielding an admitted candidate and therefore should not be interpreted as a matched final performance comparison. Responses that do not yield an admitted candidate receive no latency and are not replaced by later responses in this table.

D.4 Selected Kernel Performance

After search is complete, the selected candidate from each successful trajectory is remeasured using the timing procedure in Appendix B.

For the matched comparison between SABER and the schedule, both representations select from the same backend implementation set. We therefore evaluate final selection in two complementary ways: selected/reference latency relative to the shared finite reference, and direct pairwise latency between the implementations selected by the two representations.

Table 14 summarizes selected/reference quality. SABER selects candidates closer to the shared finite reference for both operators and under all three coding model configurations. Because both conditions use the same reference implementation, this difference reflects which implementation the coding model selects rather than differences in reachable backend quality. Because interface-specific selected/reference aggregates may contain different successful trajectory sets, their ratio need not equal the paired workload-level ratio reported in Table 15.

For completeness, Table 15 reports the pairwise selected kernel speedups from the final replay. The pairwise ratios between SABER and the matched schedule are close to one because both

Table 15: Selected kernel speedups in the experiment with ten workloads. Values above one favor the method named before "over". Parentheses report the number of complete workload pairs when fewer than the full operator set or all ten workloads are available.

Model	Ratio	SpMM	SDDMM	Overall
GPT-5.6	SABER over Matched schedule	1.0525	1.0922	1.0722
	SABER over Direct CUDA	0.9400	0.9600	0.9499
	Matched schedule over Direct CUDA	0.8931	0.8789	0.8860
GPT-6	SABER over Matched schedule	1.0459	1.0830	1.0643
	SABER over Direct CUDA	0.9100	0.9300	0.9199
	Matched schedule over Direct CUDA	0.8701	0.8587	0.8644
MiniMax-M3	SABER over Matched schedule	1.0649 (5/5)	1.0988 (4/5)	1.0798 (9/10)
	SABER over Direct CUDA	0.9800 (5/5)	1.0500 (2/5)	0.9995 (7/10)
	Matched schedule over Direct CUDA	0.9202 (5/5)	0.9556 (2/5)	0.9302 (7/10)

Table 16: Completeness of MiniMax final performance measurements. A workload is complete only when all three required trajectory replays are valid.

Interface	SpMM	SDDMM	Overall
Direct CUDA	5/5	2/5	7/10
Matched schedule	5/5	4/5	9/10
SABER	5/5	5/5	10/10

representations select from the same implementation portfolio. Differences primarily arise when the coding models select different candidate identities; small replay-timing variation can also produce ratios different from one for the same identity (Appendix E). By contrast, Direct CUDA remains a different executable space and is included only as a full-system baseline.

The independently configured SABER–SparseTIR performance comparison reported in Appendix F is separate from this matched experiment. Those configured backend measurements are not pooled with the results in this section.

D.5 Incomplete MiniMax Final Performance Results

Under MiniMax, Direct CUDA produces valid final measurements for all five SpMM workloads but only two of the five SDDMM workloads. SDDMM workloads 001, 010, and 037 do not have all three valid trajectory measurements and are therefore excluded from paired comparisons at the workload level.

The matched schedule condition has one unsuccessful trajectory, leaving one workload cell without all three required final measurements. SABER has complete measurements for all ten workloads. Pairwise performance ratios therefore use only workloads for which both required interface cells are complete; no missing value is imputed.

As a result, MiniMax comparisons involving Direct CUDA and the matched schedule may use different numbers of workload pairs. All affected ratios are reported with their pair counts.

D.6 Comparison Across Coding Models

The search pattern in the matched comparison appears under all three coding model configurations, although its magnitude depends on the model. The mean response indices for SABER are 2.40, 2.27, and 2.83 under GPT-5.6, GPT-6, and MiniMax, compared with 3.83, 3.67, and 4.24 for the matched schedule representation. Direct CUDA requires 5.60, 5.20, and 4.31 responses on average.

Reliability also remains model dependent. The GPT controlled runs have no format failures, whereas MiniMax produces 12 and 6 format failures for the matched schedule and SABER representations, respectively. Because all mapped candidates belong to the prevalidated shared portfolio, the matched

Table 17: Shared finite candidate sets in the controlled comparison.

Operator	Workloads	Candidates/workload	Points
SpMM	5	32	160
SDDMM	5	32	160
Total	10	–	320

runs contain no candidate hard failures. Direct CUDA, which generates arbitrary source code, retains substantially more candidate failures under MiniMax.

Final selection quality follows the same qualitative pattern in the matched results. SABER selects candidates closer to the shared finite reference than the matched schedule representation for both SpMM and SDDMM under all three coding model configurations. This comparison is meaningful because the two representations share candidate cardinality, candidate identity, backend code, and reference optimum.

The matched SABER–schedule study measures how the representation presented to the coding agent affects search and selection within a fixed implementation set. Direct CUDA and the independently configured SABER–SparseTIR experiments instead measure broader system behavior and reachable backend quality.

E Shared Reference Analysis

E.1 Shared Candidate Sets

The matched experiment contains ten workloads with 32 shared candidate identities per workload, giving 320 workload/candidate points. The candidate set is enumerated once and reused across coding models and agent-facing representations.

Because both controlled representations address the same implementation identities, each workload has one finite reference: the fastest correct implementation in $\mathcal{K}_w$. This reference is internal to the declared portfolio of 32 candidates and is not a global optimum over all possible GPU implementations.

E.2 Selected Candidate versus Shared Reference

Candidate identity and measured latency are reported separately. An identity match means that the coding model selected exactly the fastest candidate found by enumeration for that workload. Selected/reference latency is measured against the same reference for each workload for both controlled representations. For an interface-specific successful trajectory set $\mathcal{S}$, we report

$$R_{\mathrm{sel/ref}} = \exp\left(\frac{1}{|\mathcal{S}|}\sum_{r\in\mathcal{S}} \log \frac{L_{\mathrm{selected},r}}{L_{\mathrm{ref},w(r)}}\right).$$

The reported values measure selection quality relative to one shared reference rather than separate references specific to each interface.

E.3 Remeasurement Across Phases

Selected candidates and shared reference candidates may be timed in different measurement phases. Candidate identity is therefore reported separately from latency. When the selected identity matches the shared reference, a selected/reference ratio different from one is interpreted as timing variation across phases rather than selection regret.

When identities differ, the latency ratio can contain both selection error and measurement variation. We therefore report identity match and latency together rather than inferring one from the other.

Search metrics retain all workloads because unsuccessful trajectories are part of the observed search outcome. Bootstrap for final selection samples only complete paired workload cells. For MiniMax, the

Table 18: Selection quality relative to the shared finite reference over the 32-candidate portfolio. For each model, operator, and interface, Sel./ref. is the geometric mean of $L_{\text{selected},r}/L_{\text{ref},w(r)}$ over successful selected trajectories r for that interface. Identity-match counts use the same successful-trajectory set. These interface-specific aggregates are not paired across interfaces.

		SpMM		SDDMM	
Model	**Interface**	**Sel./ref.**	**ID match**	**Sel./ref.**	**ID match**
GPT-5.6	Matched schedule	1.082	10/15	1.137	7/15
	SABER	1.028	13/15	1.041	12/15
GPT-6	Matched schedule	1.071	11/15	1.122	8/15
	SABER	1.024	14/15	1.036	13/15
MiniMax-M3	Matched schedule	1.115	8/15	1.068	6/14
	SABER	1.047	12/15	1.063	11/15

Table 19: Geometric mean latency of the shared finite reference.

	SpMM (μs)	**SDDMM (μs)**
Shared reference	8.74	6.21

comparison between SABER and the matched schedule therefore contains nine complete workload pairs; resampling preserves the operator composition of this set of complete pairs. The interface-specific aggregates in Table 18 are not divided to form the paired final-selection statistic. Table 20 recomputes that statistic directly over workload cells that are complete for both controlled interfaces.

E.4 Reference Latencies

Across the five SpMM workloads, the shared finite reference has a geometric mean latency of 8.74 μs. Across the five SDDMM workloads, the corresponding value is 6.21 μs.

Because both controlled representations address the same implementation set, there is one reference latency per workload rather than a reference specific to each interface. These values characterize the shared candidate portfolio and are not lower bounds on globally achievable GPU performance. The controlled study therefore separates two questions: how effectively each agent-facing representation guides selection within a fixed implementation set, and how fast that shared implementation set is in absolute terms.

E.5 Uncertainty of the Matched Comparison

We quantify uncertainty with a paired stratified workload bootstrap using 10,000 bootstrap samples and percentile 95% confidence intervals. Quantiles use linear interpolation. Resampling is performed at the workload level, so the three trajectories associated with a sampled workload remain grouped; timing samples within a workload are not treated as independent observations.

Search metrics. For each search-metric bootstrap draw, we independently resample five SpMM workload identities and five SDDMM workload identities with replacement. A sampled workload carries all three trajectories from both controlled interfaces. For ΔBest, we pool the response-to-best values of all trajectories that eventually produce a correct candidate within the resampled workloads, following the definition in Appendix B.8, and compute the matched-schedule mean minus the SABER mean. For ΔAt-2, unsuccessful or not-yet-admitted prefixes remain non-attaining observations; we pool the trajectory-level attainment outcomes from the resampled workloads and compute the SABER attainment rate minus the matched-schedule rate. Positive values therefore favor SABER.

Final selection. Final-selection uncertainty is computed only from workload cells that are complete for both controlled interfaces. GPT-5.6 and GPT-6 therefore resample five SpMM and five SDDMM paired workload cells, whereas MiniMax resamples five SpMM and four SDDMM paired workload

Table 20: Paired bootstrap uncertainty by workload for the matched comparison. Brackets report percentile 95% confidence intervals from the paired workload-stratified bootstrap; quantiles use linear interpolation.

Model	ΔBest	ΔAt 2	SABER over matched
GPT-5.6	1.43 [0.72, 2.16]	+23.3 [10.0, 36.7] pp	1.072 [1.018, 1.129]
GPT-6	1.40 [0.69, 2.12]	+33.3 [16.7, 50.0] pp	1.064 [1.015, 1.118]
MiniMax-M3	1.41 [0.42, 2.38]	+40.0 [20.0, 60.0] pp	1.080 [1.009, 1.115]

Table 21: Anytime attainment against the shared reference in the matched experiment. R2 and R8 denote attainment after responses two and eight, respectively.

		R2 attainment			R8 attainment		
Model	**Interface**	**1%**	**5%**	**10%**	**1%**	**5%**	**10%**
GPT-5.6	Matched	30.0	46.7	60.0	60.0	76.7	86.7
	SABER	60.0	73.3	83.3	86.7	93.3	96.7
GPT-6	Matched	36.7	53.3	63.3	66.7	80.0	90.0
	SABER	66.7	76.7	86.7	90.0	96.7	100.0
MiniMax-M3	Matched	23.3	36.7	46.7	50.0	66.7	76.7
	SABER	46.7	60.0	70.0	80.0	90.0	96.7
Aggregate	Matched	30.0	45.6	56.7	58.9	74.4	84.4
	SABER	57.8	70.0	80.0	85.6	93.3	97.8

cells. Within each operator stratum, complete paired workload cells are sampled with replacement. For each draw, the paired final-selection statistic is the geometric mean SABER speedup over the matched schedule, $L_{\mathrm{Matched}}/L_{\mathrm{SABER}}$, over the resampled workload cells; values above one favor SABER. The reported confidence intervals are the 2.5th and 97.5th percentiles of the corresponding bootstrap distributions.

E.6 Anytime Progress Against the Shared Reference

Because the complete portfolio of 32 candidates is enumerated independently of agent search, we additionally evaluate progress against a fixed reference for each workload. For trajectory r on workload w, let $c_{r,j}$ denote the admitted candidate identity produced by response j, and let $\ell_w(c)$ be its latency from the enumeration of the finite candidate set. We define

$$Q_{r,w}(t) = \frac{\min_{j \le t} \ell_w(c_{r,j})}{\ell_w^\star},$$

where $\ell_w^\star$ is the fastest correct latency in the shared portfolio. A prefix with no admitted candidate is treated as not attaining any threshold.

The metric using the shared reference shows the same ordering as the retrospective search metrics. At response two, aggregate attainment within 10% of the shared finite reference is 80.0% for SABER versus 56.7% for the matched schedule; the corresponding 5% rates are 70.0% and 45.6%. By response eight, the corresponding 10% attainment rates rise to 97.8% and 84.4%. Thus, the earlier progress observed with SABER is not an artifact of defining the target relative to each trajectory's own final discovered best.

F Comparison of Independently Configured SABER and SparseTIR Systems

This section evaluates independently configured SABER and SparseTIR systems. These experiments are separate from the matched abstraction control in Sections 4.4 and Appendix D. The two systems use different candidate domains and backend implementations, so the results characterize configured backend quality rather than isolate the agent-facing representation.

Table 22: Candidate domains in the independently configured SABER–SparseTIR comparison.

Operator	SparseTIR	SABER
SpMM	1	2
SDDMM	72 or 96	1

Table 23: Selected kernel SABER speedup over independently configured SparseTIR. Each operator uses five workload pairs.

Model	SpMM	SDDMM	Overall
GPT-5.6	1.9374	0.7349	1.1932
GPT-6	1.9260	0.7216	1.1789
MiniMax-M3	1.9937	0.7523	1.2247

SparseTIR uses upstream commit `9b3f8436de7724a40899c82e7c42c40fc33e0285` together with the local SM86 compatibility configuration used in our runs. The runtime reports TVM 0.9.dev0. These measurements should be interpreted as results for this specific adapter and software version, not for every possible SparseTIR configuration.

F.1 Configured Candidate Domains

The independently configured domains differ substantially in size.

These domains are not used for the search claim from the matched comparison; they characterize reachable performance of the configured systems.

F.2 Configured Performance Across Three Models

Table 23 reports selected kernel performance for the independently configured systems. All values are SABER speedup over SparseTIR; values above one favor SABER.

The operator direction is consistent across the three coding model configurations: the configured SABER backend is faster on the five SpMM workloads, while the configured SparseTIR backend is faster on the five SDDMM workloads. These ratios combine differences in candidate domains, backend code, and model selections and are therefore not interpreted as a causal effect of abstraction level.

F.3 Candidate-Set Controls

Figure 4(b) reports a separate SABER–SparseTIR measurement on the same five CSR SpMM and five CSR SDDMM workloads. The selected condition reports the implementations selected in that measurement and is distinct from the model-specific configured selections reported in Table 23. The two sets of results are measured separately and are not pooled. The configured-reference condition reports the best measured correct implementation available within each configured candidate domain, while the one-candidate condition retains exactly one candidate per workload and interface using the deterministic ordering described below.

For the selected implementations, SABER speedup over SparseTIR is 2.1750× for SpMM and 0.6797× for SDDMM. The corresponding configured-reference speedups are 2.1703× and 0.6675×. With exactly one candidate per interface, the speedups are 1.2354× and 0.6383×, respectively.

F.4 Single Candidate Control

To test whether candidate count alone explains the difference in final latency in the independently configured comparison, we additionally measure exactly one candidate per workload and interface. Candidate membership is fixed before timing by placing normalized candidate identities in a deterministic order and retaining only the first candidate for each workload/interface pair. The ordering does not use measured latency or correctness.

Table 24: Configured backend SABER speedup over SparseTIR under several measurement conditions. The rows are reported separately because their candidate sets and timing sessions differ.

Comparison	SpMM	SDDMM
Selected	2.1750	0.6797
Configured reference	2.1703	0.6675
One-candidate	1.2354	0.6383

With one candidate per interface, SABER speedup over SparseTIR is 1.2354 for SpMM and 0.6383 for SDDMM. The SpMM advantage becomes smaller than in the larger configured comparison but remains above one; SDDMM remains below one.

This experiment controls candidate count for final measured performance only. It does not reproduce the search process with a domain of equal size and does not isolate abstraction level from backend implementation quality.

F.5 Fixed Prefix Measurements

The single candidate result is part of a larger fixed prefix measurement. For each workload/interface pair, candidates are kept in the same deterministic ordering while the retained prefix is increased without using measured performance to choose membership.

At prefix length one, both interfaces contain exactly one candidate per workload. At larger requested prefix sizes, the domains no longer remain equal: SABER has only two SpMM candidates, SparseTIR has only one SpMM candidate, and SABER has only one SDDMM candidate. Larger prefixes therefore should not be interpreted as comparisons with equal cardinality.

Across the complete fixed prefix measurement, there are forty correct workload/configuration points and 800 retained timing sample means. Each sample mean averages twenty launches after five warmups. The single candidate result is a subset of these measurements rather than forty independent workloads.

F.6 Interpretation

The configured backend controls answer a different question from the matched study. The matched experiment asks how two representations guide selection within the same implementation portfolio. The experiments in this section ask how the independently configured SABER and SparseTIR systems perform with their configured candidate domains and backend implementations.

The single candidate result makes candidate count less plausible as the sole explanation for the difference in configured final performance, but the two remaining candidates are still different implementations. The result therefore does not isolate the abstraction boundary itself.

G Replay Across Architectures

Hardware environments. The cross-architecture replay experiments use NVIDIA A100 and H200 compute nodes on the NCSA Delta system. Table 25 summarizes the relevant hardware configurations. The A100 measurements use a 4-GPU node with 40 GB A100 SXM GPUs (Compute Capability 8.0), while the H200 measurements use an 8-GPU node with 141 GB H200 SXM GPUs (Compute Capability 9.0). The A100 node is hosted by a 64-core AMD Milan CPU with 256 GB of system memory; the H200 node uses two Intel Xeon Platinum 8558 CPUs with 96 total cores and 2,048 GB of system memory. Hardware multithreading is disabled on both node types. All reported kernel measurements use a single GPU; the remaining GPUs on the node are not used by the benchmark.

Replay methodology. We evaluate how the configured SABER and SparseTIR implementations transfer across additional NVIDIA GPU architectures. Starting from the candidate identities selected by GPT-5.6 on RTX 3090, we recompile and replay the same implementations on A100 and H200. The workload identities, input values, numerical contract, and complete-device-operation timing

Table 25: Hardware used for the cross-architecture replay experiments.

Specification	A100 node	H200 node
GPU	NVIDIA A100 (SXM)	NVIDIA H200 (SXM)
Compute capability	8.0	9.0
GPUs per node	4	8
GPU memory	40 GB	141 GB
Host CPU	AMD Milan	Intel Xeon Platinum 8558
CPU sockets	1	2
CPU cores / node	64	96
Host memory	256 GB	2,048 GB
Hardware threads / core	1 (SMT off)	1 (Hyper-Threading off)

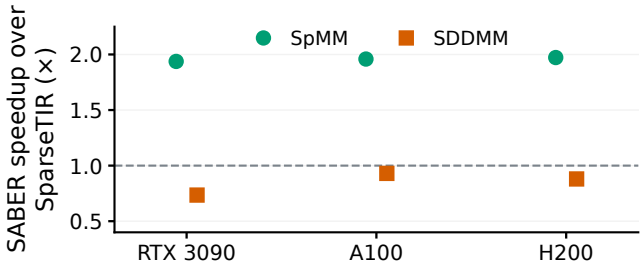


Figure 7: Configured GPT-5.6-selected implementations evaluated on RTX 3090, A100, and H200.

methodology follow Appendix B. Keeping the selected implementation identities fixed allows the comparison to characterize cross-architecture behavior without introducing a different selection outcome.

The operator-level performance direction is consistent across the three measured GPUs. For SpMM, SABER remains faster than configured SparseTIR, with speedups of 1.9374×, 1.9588×, and 1.9723× on RTX 3090, A100, and H200, respectively. For SDDMM, configured SparseTIR remains faster, although the gap varies across targets: SABER speedup over SparseTIR is 0.7349×, 0.9323×, and 0.8816×, respectively. The corresponding geometric means across the ten workloads are 1.1932×, 1.3514×, and 1.3186×.

Thus, for these fixed GPT-5.6-selected implementations, the configured systems preserve the same operator-level performance direction across all three GPUs.

H Additional Measurements

H.1 Generation Cost

Table 27 reports mean output length and generation time per response in the controlled experiment with ten workloads. Generation time is separate from GPU execution time and excludes compilation and correctness checking, while token count measures response length rather than a fixed amount of computation. We therefore compare generation burden across interfaces only within the same coding model configuration, not across model families.

H.2 Search and Selection Without an LLM

The SABER interface does not require an LLM. We evaluate fixed, heuristic, and decision-tree selection on 72 additional workloads using previously measured finite candidate sets. No new model calls or GPU measurements are introduced in this experiment. Each policy selects among existing SABER candidates, and selection quality is evaluated against the fastest correct candidate in the corresponding finite set. Figure 8 shows the resulting selection quality per workload.

Table 26: Configured GPT-5.6-selected implementations evaluated on RTX 3090, A100, and H200. Values report SABER speedup over independently configured SparseTIR.

GPU	SpMM	SDDMM	Overall
RTX 3090	1.9374×	0.7349×	1.1932×
A100	1.9588×	0.9323×	1.3514×
H200	1.9723×	0.8816×	1.3186×

Table 27: Generation burden per response in the experiment with ten workloads.

Model	Interface	Tokens	Seconds	Responses
GPT-5.6	Direct CUDA	928.2	21.32	240
	Matched schedule	348.0	10.90	240
	SABER	148.0	7.20	240
GPT-6	Direct CUDA	948.4	35.35	240
	Matched schedule	320.0	14.50	240
	SABER	142.0	9.60	240
MiniMax-M3	Direct CUDA	2011.2	17.45	240
	Matched schedule	620.0	10.00	240
	SABER	340.0	7.80	240

H.3 Additional Operator Implementations

The executable backend also contains implementations for SpMV, reductions, SpGEMM, and sparse attention. Figure 9 shows the retained implementation measurements for these operators.

H.4 SDDMM Backend Realizations

A public SABER action does not uniquely determine its physical GPU implementation. We therefore measured several backend realizations for the same five CSR SDDMM workloads.

The measurements are divided into four groups. Table 28 shows the size of each group.

All 105 workload/variant points pass correctness. The measurements include both fast and slow implementations; variants are not removed based on their latency.

Figure 10 shows the complete results. Each panel is normalized independently to the fastest implementation for each workload within that measurement group. The numerical values and colors are therefore meaningful within a panel and should not be compared directly across different panels.

A separate matched measurement compares the original direct realization with the selected private realization on the same five workloads. The selected realization provides a 1.3099× geometric mean speedup over the direct implementation.

H.5 SpGEMM Backend Change

We also changed the private SpGEMM implementation without changing its public SABER action. The newer lowering combines symbolic and numeric work and emits COO output directly from a row-owned implementation.

Source inspection shows fewer kernel launch boundaries and different intermediate storage than the earlier implementation.

The comparison contains two SpGEMM workloads with three repetitions each. The newer private implementation improves SABER latency by 3.3578× geometric mean. Over the same workloads, SABER speedup over Direct CUDA increases from 0.2764 to 0.9282.

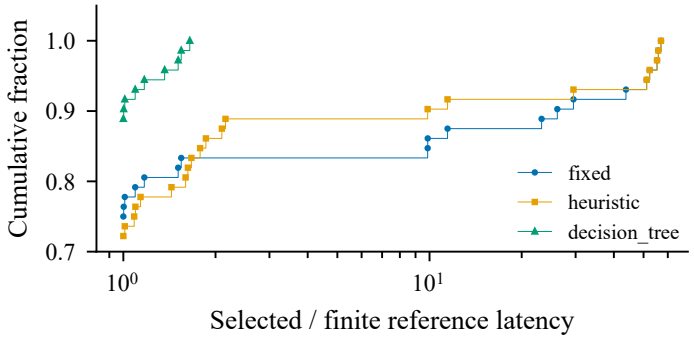


Figure 8: Offline policy selection on 72 workloads using the previously measured finite candidate sets. The policies select among existing SABER candidates; no new model calls or GPU measurements are introduced for this comparison. Selection quality is evaluated against the fastest correct candidate available in the corresponding finite set.

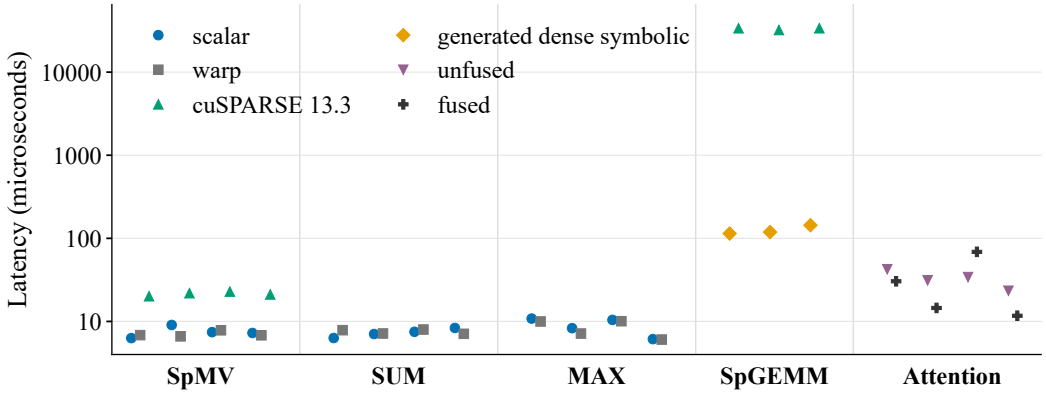


Figure 9: Additional GPU implementations for SpMV, SUM reduction, MAX reduction, SpGEMM, and sparse attention. The figure documents additional executable operator paths and realization choices; it is not used as a held-out vocabulary result. Latency is shown on a logarithmic scale.

The workload with a heavy tail still favors Direct CUDA after this change. The result therefore shows that moving an implementation choice below the public boundary can recover part of a performance gap, but does not imply that every gap can be eliminated in the backend.

The public semantic state is unchanged, and no additional agent call or new public action is introduced when replacing the backend implementation.

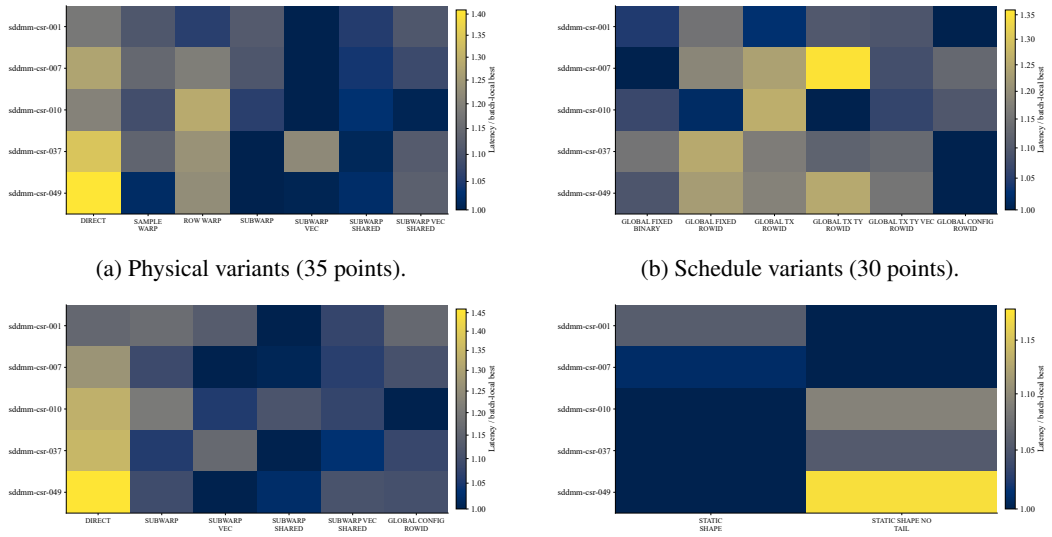
H.6 Selecting Among Private Realizations

The SDDMM backend contains a private portfolio of eight realizations that differ in cooperation width, vectorization, row mapping, shared-memory staging, and shape specialization. We evaluate two deterministic, compiler-private selection rules over this fixed portfolio. Neither selector is learned: both use hard-coded thresholds over workload features such as the reduction dimension, row-density statistics, empty-row fraction, and, for the revised rule, target architecture. The measured portfolio contains eight realizations, although each deterministic rule can select only a rule-specific subset of four realizations under its current decision branches.

The initial selector was evaluated during Stage12D on five calibration workloads and six additional workloads. Results from all eleven Stage12D workloads were subsequently used as development evidence for the revised selector; the six-workload subset therefore should not be interpreted as held

Table 28: Private SDDMM implementation measurements. Each group uses the same five workloads. Variant counts refer to the implementations included in that measurement group.

Implementation group	Variants	Workloads	Points
Physical variants	7	5	35
Schedule variants	6	5	30
Native realizations	6	5	30
Shape specialization	2	5	10
Total			**105**



(a) Physical variants (35 points).

(b) Schedule variants (30 points).

(c) Native realizations (30 points).

(d) Shape-specialized variants (10 points).

Figure 10: SDDMM backend implementations measured on five workloads. The four groups contain 105 workload/variant points in total, all of which pass correctness. Each panel is normalized independently to the minimum latency for each workload within that group. Slow variants are retained rather than filtered.

out for the revised rule. In particular, the revision explicitly used a failure case from this subset to adjust the private realization rule.

We then compare the initial and revised selectors on sixteen additional Stage12E workloads over the same eight private realizations. Repository records designate this set as latency-held-out during revised-selector development: the workload specifications and revised-selector source were frozen before Stage12E timing, and the current selector, workload, and measurement hashes match the archived freeze records. These artifacts support the recorded separation between selector development and Stage12E timing, although they do not independently reconstruct the complete historical inspection sequence.

For each workload, the reference is the fastest measured correct realization among the eight private variants. Table 29 reports selection regret, defined as selected/reference latency minus one. On the sixteen Stage12E workloads, the revised rule reduces geometric-mean excess from 7.57% to 4.64% and worst-case regret from 106.08% to 18.12%. It does not improve every metric: median regret increases from 1.07% to 1.55%, and the fraction of workloads within 10% of the reference decreases from 87.5% to 75.0%.

All 128 workload/variant combinations pass correctness. The differences therefore reflect private-realization selection rather than missing executable variants.

Table 29: Selection among private implementations on sixteen additional workloads. Regret is selected/reference minus one. "Within 10%" counts workloads whose selected implementation is no more than 10% slower than the fastest measured private variant.

Metric	Initial selector	Revised selector
GM excess	7.57%	4.64%
Median regret	1.07%	1.55%
Worst regret	106.08%	18.12%
Within 10%	87.5%	75.0%

Table 30: Representative implementation footprint. LOC excludes blank and lines containing only comments. Percentages are relative to the components reported in this table.

Component	LOC	Share
Public action / state schema	399	9.8%
Verifier / canonicalization	515	12.7%
Shared backend infrastructure	885	21.8%
SpMM backend	1,348	33.2%
SDDMM backend	838	20.7%
SpGEMM backend	71	1.8%
Total	4,056	100%

H.7 Implementation Footprint

To quantify the engineering footprint around the abstraction boundary, Table 30 reports implementation LOC for representative interface, verification, shared backend, and operator-specific components. Counts exclude blank and lines containing only comments and are intended as an indicator of implementation size rather than a measure of human engineering effort. Shared backend infrastructure includes common planning, dispatch, runtime, and sparse data support used across operator-specific paths.

The implementation contains 399 LOC defining the public action and state schema, 515 LOC for verification and canonicalization, and 885 LOC of shared backend infrastructure. The operator-specific implementations contain 1,348 LOC for SpMM, 838 LOC for SDDMM, and 71 LOC for the SpGEMM paths used in the reported comparison.

The footprint confirms that SABER shifts rather than removes implementation complexity. Of the 4,056 LOC reported here, operator-specific backends account for 2,257 LOC, in addition to 885 LOC of shared infrastructure. This complexity can nevertheless be reused below a fixed public action: Sections H.4 and H.6 evaluate several SDDMM realization families and a private portfolio of eight variants whose cooperation, vectorization, storage, and scheduling choices do not introduce new public actions.

The SpGEMM experiment provides a second example. Replacing the private lowering improves SABER latency by 3.3578× while leaving the action visible to the coding agent unchanged. Thus, SABER does not eliminate implementation engineering; it separates that engineering from agent search so that backend realizations can be reused and improved independently.

H.8 Compilation and Cache Behavior

Compilation and host-side setup are not included in the GPU latency numbers reported in the main experiment. We separately measured several parts of this cost.

A cache experiment uses three workloads and three repetitions, giving 27 correct measurements across cold, disk-warm, and memory-warm conditions. Relative to the cold path, disk-warm wrapper time improves by 11.89–12.28×. This is a host-side wrapper result rather than a GPU kernel speedup.

In a separate prepared plan measurement, total host execution time ranges from 5.707 to 6.981 seconds, while the measured compiler processing portion ranges from 1.142 to 1.916 milliseconds.

Plan construction, portfolio selection, and input preparation occur before the compiler processing timer. Compilation of generated code and benchmark process startup dominate the larger wrapper time. These measurements are kept separate from the device operation latency used for kernel comparisons.

H.9 Tensor Core Numerical Details

A separate Tensor Core comparison uses thirteen aligned BSR16 SpMM workloads. Inputs are FP16, while accumulation and output are FP32. The correctness requirement is

$$\mathrm{atol} = \mathrm{rtol} = 5 \times 10^{-3}.$$

All thirteen workloads pass this contract. The Tensor Core implementation provides a 12.839× geometric mean speedup over the matched FP16 CUDA core implementation. The recorded instruction trace for this run contains `HMMA.16816.F32` instructions on SM86. We use this instruction record only to confirm Tensor Core execution for this particular implementation and GPU. Use of a Tensor Core API in source code would not establish which instructions execute.

This result uses a different numerical contract from the FP32 experiment with ten workloads. It therefore does not justify changing the precision or tolerance used in the main comparison. This Tensor Core result characterizes the RTX 3090 implementation and is reported separately from the configured backend results across architectures in Appendix G.