SPARSEKERNELBENCH: A BENCHMARK FOR SPARSE GPU COMPUTATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Sparse GPU performance depends on both the number and locations of nonzeros, requiring evaluation across input structures. We introduce **SparseKernelBench**, with 41 tasks across seven sparse representations, 533 performance cases and 637 correctness-only cases. Each task specifies input/output requirements and input families that a program must support. Four large language models generate programs without execution feedback for every task. Haiku produces no custom kernels. Programs from the other three models outperform the fastest tested library baseline, including compiled framework compositions, on 36.6–54.2% of the 533 performance cases. Requiring every task check to pass gives 36.0–44.3%, with the same denominator. Three of 103 programs that pass every performance-input check fail dedicated correctness-only inputs. At fixed dimensions, nonzero count and feature width, the fastest generated sampled dense–dense multiplication program takes $1.98\times$ the fastest tested baseline's runtime on uniform inputs but $0.54\times$ on skewed inputs. Changing the baseline formulation also reverses whether programs outperform the library on 29–64 cases per model. These results show why sparse evaluation must establish the input domain, numerical requirements and tested comparator behind a performance claim.

1 INTRODUCTION

Sparse GPU performance depends on both how many nonzeros an input has and where they occur. Dense multiplication has a regular computation specified by dimensions and layouts. Sparse multiplication follows indices, so matrices of the same shape and density can differ in memory access, data reuse and work per thread. One pattern cannot establish performance across a task's inputs.

Curating operations is therefore only one axis of a sparse benchmark. The second is selecting application-motivated input patterns for each operation. Uneven graph degrees produce unequal row workloads, spatial discretizations produce local connections, and triangular solves follow row dependencies. These structures require different scheduling and data reuse. Changing values or dimensions within one family does not exercise the others. KernelBench (Ouyang et al., 2025a) curates operations from PyTorch workloads, and robust-kbench (Lange et al., 2025) extends testing across configurations. SuiteSparse (Davis & Hu, 2011) and DLMC (Gale et al., 2020) document application-derived sparse matrices. A matrix collection alone does not specify an operation's required inputs, output correctness or library comparators.

SparseKernelBench makes the input domain, correctness requirements and tested baselines explicit. Each of its 41 tasks pairs an operation and input/output requirements with application-motivated input families, spanning seven sparse representations and computations from primitives to model components. One program must handle every input of its task. Dedicated correctness tests exercise conditions such as empty structures and irregular dimensions that may expose errors but contain too little work for useful timing. We compare against the fastest tested baseline on each input and report both per-case performance and performance restricted to programs that pass every task test. Failed and unmeasured performance cases remain in both scores' denominators.

Experiments with programs generated without execution feedback demonstrate three ways these choices change an evaluation's conclusion. At fixed dimensions, nonzero count and feature width, changing sparse structure makes a library implementation preferable to the best generated sampled dense–dense multiplication (SDDMM) program on uniform inputs, but the generated program

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

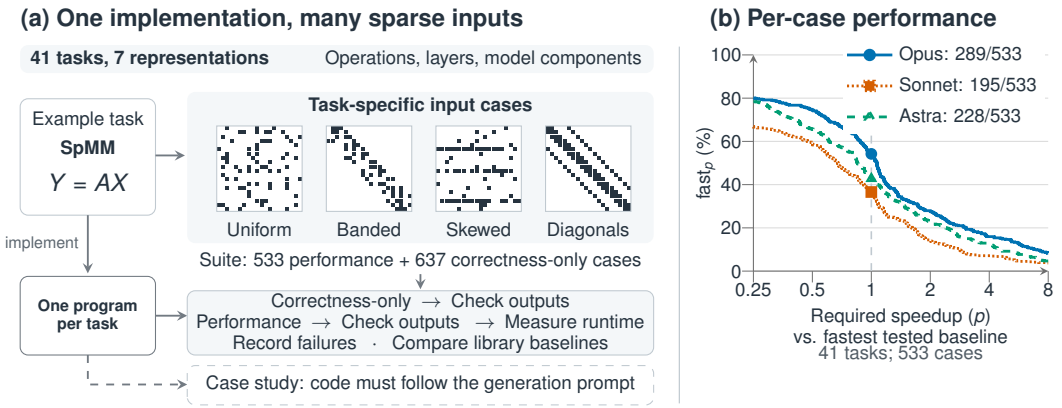


Figure 1: **SparseKernelBench evaluates one program per task across input families.** SpMM: sparse matrix–matrix multiplication. (a) Synthetic inputs in the study. Sketches are illustrative. (b) Primary comparison: Core and large suites. fast_p counts cases with a prompt-compliant program and valid speedup above p against the fastest tested baseline, over all 533 performance cases. Legend counts use $p = 1$. Failed and unmeasured cases remain in the denominator. Table 1 also reports scores requiring every task test to pass.

preferable on skewed inputs, using the faster tested cuSPARSE or PyTorch sampled matrix multiplication baseline. Three programs pass every performance-input check yet fail dedicated correctness inputs. Finally, 29 Opus, 64 Sonnet and 54 Astra comparisons favor the program against the default baseline but favor the fastest tested baseline. Broader structural coverage reveals changes in relative performance missed by uniform-only testing, even when the fastest program on average stays the same. The benchmark, protocol and evidence are detailed in Sections 3 to 6.

2 Related work

Sparse datasets and benchmarks. SuiteSparse (Davis & Hu, 2011) and DLMC (Gale et al., 2020) preserve application-derived patterns. SparseBench (Dongarra et al., 2001) evaluates solvers, preconditioners and storage schemes. SparseKernelBench combines task-specific error tolerances, multiple input families and results showing which inputs each implementation supports.

Sparse libraries and compilers. cuSPARSE (NVIDIA, 2025), Sputnik (Gale et al., 2020) and GE-SpMM (Huang et al., 2020) optimize sparse operations for different workloads. TACO (Kjolstad et al., 2017) and SparseTIR (Ye et al., 2023) specialize code to representations. Root et al. (2024) fuse shape operations with sparse computation. Scorch (Yan et al., 2026) autoschedules loop orders, tiling and intermediate formats on CPUs. NVIDIA's 2:4 sparsity (Mishra et al., 2021) enables dedicated hardware. Our input families test which inputs different implementations support and how their performance varies across those inputs.

Evaluating generated kernels. KernelBench (Ouyang et al., 2025a) evaluates PyTorch-module replacements. Robust-kbench (Lange et al., 2025) varies configurations and unseen shapes. KernelBench-Verified (Zhang et al., 2026) changes baseline precision and tests correctness across distributions. MSKernelBench (Han et al., 2026) includes sparse products, and SparseDitto (Li et al., 2026) specializes to a matrix, operator and GPU. Our emphasis is on multiple sparse input families under one task definition, records of baseline support and scores that include failed cases. We apply KernelBench's score to performance *cases*, rather than tasks, and require valid timing and compliance with the generation prompt as well as passing the numerical output checks. Our prompts require numerical work in generated kernels, whereas KernelBench's February 2025 preprint permits selective operator replacement (Ouyang et al., 2025b). These differences preclude direct comparison of success percentages.

3 SPARSEKERNELBENCH

A task defines an operation and its input/output requirements. A case is one input. Models receive these requirements, a function signature, tolerances and a reference implementation. One Python module handles all task cases and may launch multiple GPU kernels. Every timed call includes input-dependent conversion, index construction and intermediate computation. This preparation is not amortized across uses of one pattern (Section 4). Fresh subprocesses contain accidental failures.

Multiplication and embedding reductions involve uneven work, tested by balanced and skewed rows. Sparse–sparse products construct indices and values: empty, banded and stored zero cases test the output pattern and result. Triangular solves follow row dependencies, so diagonal and bidiagonal cases test independent rows and sequential chains. Sparse convolution locates neighboring coordinates. Its inputs vary occupancy, channel widths and boundaries. The 41 tasks comprise 28 single operations, seven composite layers and six model components. A graph convolution layer computes $Y = \mathrm{ReLU}(A(XW) + b)$ from normalized adjacency A, features X, weights W and bias b. The transformer block includes sparse attention, projections, residual connections, layer normalization and a feed-forward network. Only final outputs are required, allowing fusion (Appendix A.2).

Six storage formats are compressed sparse row (CSR) and column (CSC), coordinate (COO), block CSR (BSR), ELLPACK (ELL) and diagonal (DIA). The seventh representation, 2:4 sparsity, stores two values and their positions per four consecutive columns of each row. Representation variants are separate tasks: SpMV groups contributions by output row in CSR but by input column in CSC, requiring combination across columns.

Choosing input patterns. Three broad classes motivate our selection. *Networks with uneven degrees*, including social and information networks, often have heavy-tailed degrees, sometimes approximated by power laws, and short paths between vertices (Leskovec et al., 2007). In their adjacency matrices, a few highly connected vertices create long rows among many short ones. Skewed inputs test this imbalance in SpMV, SpMM and graph layers, while equal-length rows provide a balanced comparison.

Spatial discretizations connect neighboring physical locations. Heat-diffusion grids have roughly uniform degree and paths growing with grid extent. Unstructured meshes and rigid-body contact networks have irregular but local connectivity. Bounded neighbor counts make nonzeros grow linearly with matrix dimension and density decrease. Banded and stencil inputs test locality under a spatial ordering. Our fixed-offset generators clip at matrix bounds, without enforcing spatial grid boundaries (Appendix C.8).

Pruned weights come from sparsifying dense neural-network matrices and are often denser than discretization matrices (Gale et al., 2020). At a fixed retained fraction, density stays constant with size, so wider matrices have more nonzeros per row. Multiplication tasks use uniformly scattered entries, dense blocks and 2:4 structure where the representation permits (Mishra et al., 2021).

Figure 2 groups these properties into categories that can overlap, as with banded triangular matrices. *Other* includes equal-length rows, blocks, sliding windows, required diagonals, tensor coordinates and 2:4 structure. These families test selected properties, not entire application graphs, meshes or learned weight distributions. Section 6.3 tests their effects at fixed size and feature width.

Test inputs. The synthetic study has 533 performance cases for checking and timing across small, medium and large resource tiers, and 637 correctness-only cases, many too small for informative timing (Appendices A.3 and A.4). Empty structures test boundary behavior, while value distributions, stored zeros, cancellation and conditioning test numerical requirements. Pre-filled output buffers expose unwritten elements. Repeated and transformed inputs test for stale results (Appendix B.3). SpMV and SpMM also use scientific matrix patterns (Section 5). CSR SpMV means of 4, 16 and 64 nonzeros per row vary reduction work, while SpMM widths 32 and 129 test aligned and partially filled tiles. Square, tall, wide and irregular dimensions vary work distribution and boundary handling. Index widths vary memory traffic and library support, and resource tiers vary working-set size. The aggregate weights cases chosen for coverage of these properties, independently of their frequency in applications. Appendix C.13 quantifies overlap and gaps with the scientific matrices.

Operation family	Tasks	Input patterns						Performance	Correctness only
		Unif.	Band.	Skew.	Diag.	Tri.	Other		
Single operations	**28**							**425**	**451**
SpMV	5	●○	●○	●○	●○		●○	76	75
SpMM	7	●○	●○	●○	●○		●○	140	123
SDDMM	1	●○	●○	●○			●○	18	16
SpGEMM / Gram	2	●○	●○	●○		○	●○	13	32
Sparse softmax	1	●○	●○	●○			●○	15	16
Triangular solves	3					●○	●	39	45
IC(0) / ILU(0)	2					●○	●○	25	28
Embedding bags	4	●○		●○			●	54	67
Sparse convolution	1						●○	10	16
MTTKRP	2						●○	35	33
Composite layers	**7**							**59**	**101**
GNN layers	3	●○	●○	●○			●○	22	42
Sparse attention	3	●○	●○	●○		●○	●○	30	45
MoE feed-forward	1	●○		●○			●○	7	14
Model components	**6**							**49**	**85**
GNN models	2						●○	9	33
Transformer block	1						●○	4	16
MoE block	1	●○		●○			●○	12	11
Recommender block	1	●○		●○			●○	12	13
Sparse conv block	1						●○	12	12
Total	**41**							**533**	**637**

● Performance cases present ○ Correctness-only cases present

Figure 2: **Task coverage and the study's synthetic inputs.** SpMV/SpMM: sparse matrix–vector/matrix–matrix multiplication; SDDMM: sampled dense–dense matrix multiplication; SpGEMM: sparse–sparse matrix multiplication; IC(0)/ILU(0): incomplete Cholesky/LU factorization with zero fill; MTTKRP: matricized tensor times Khatri–Rao product (a tensor contraction); GNN: graph neural network; MoE: mixture of experts. Pattern columns denote uniform (Unif.), banded (Band.), skewed (Skew.), diagonal (Diag.) and lower-triangular (Tri.) inputs. Other includes balanced row lengths. Filled/open dots mark performance/correctness-only tests. Blank cells contain no tests. Counts and bars describe study inputs. Appendix A.3 gives coverage by task and generator.

Example task: CSR SpMV. For row offsets r, column indices c, values a and dense vector x, return $y_i = \sum_{k=r_i}^{r_{i+1}-1} a_k x_{c_k}$. Offsets are nondecreasing from zero to the length of a. Columns strictly increase within rows, without duplicates. Values are finite float32, and indices are int32 or int64. Outputs for empty rows must be zero, inputs unchanged, and outputs within the task's numerical tolerance.

Baselines. The 79 baseline adapters use direct cuSPARSE calls, individual PyTorch sparse operations, conversion followed by an operator, or PyTorch compositions. All 32 eager compositions also have `torch.compile` variants. API and algorithm variants can share backends, so adapter counts are not library counts. We report whether a compiler variant records graph capture, which may cover only part of an operation. Each implementation is checked on each input. The *default baseline* is the first passing implementation in a fixed priority order, excluding compiled variants. The *fastest tested baseline* has the lowest runtime among tested eager or compiled implementations passing the output checks and timing requirements (Section 4). Speedup is baseline time divided by candidate time, relative to this tested set. Appendix A.5 records implementations, compilation settings, setup costs and support restrictions. The 2:4 cases use float32, which the tested semi-structured adapter does not support. Their comparisons use dense or CSR alternatives and do not measure the specialized sparse tensor-core path.

Validation. Independent tests compare baselines against float64 references. The evaluator detects the intended defect in 114 of 115 deliberately faulty programs. For calibrated model-component tolerances, all 472 held-out outputs pass. Adding one to the first element of each reference output is rejected on all 59 tested inputs in both the CPU and GPU audits. These independently test the evaluator (Appendices D.1 to D.3). Appendix B.3 specifies the checks applied to candidates.

4 EVALUATION PROTOCOL

Timing. We report median CUDA-event latency after subtracting a shared empty-event overhead estimate. Launch delays remain included. If the estimate exceeds half the candidate or baseline median, or subtraction would make that median nonpositive, we use the corresponding raw median. Without a measured overhead estimate, both sides use their raw medians. Candidate and baseline samples use the same cache treatment. Repeated measurements and groups of timed calls check stability and overhead, and diagnostic flags accompany each speedup. We exclude a timing when the host-synchronized median exceeds four times the CUDA-event median, because some work may have occurred outside the event interval. Both medians must be finite and positive. Measurement-selection rules apply equally to every model. Appendix B.2 give the checks and identify which recorded attempt supplies each result.

Work inside the timer. Each timed call starts from sparse component tensors and any dense operands already on the GPU. Both candidates and baselines must perform the conversion, index construction and intermediate computation needed to produce outputs inside the timer. Candidate preparation receives only public metadata, such as shapes and dtypes. Allocation and workspace setup from this metadata may precede timing. Baselines also validate inputs and create sparse descriptors before timing (Appendix A.5). Thus we measure input-dependent preparation on every call, without amortizing it across repeated use of a fixed sparsity pattern.

Performance score. *Source review* checks compliance with the prompt's implementation rules, separately from output testing (Section 5). For N performance cases, define

$$\text{fast}_p = \frac{1}{N} \sum_{i=1}^{N} 1[E_i \ \wedge \ s_i > p] \,, \tag{1}$$

where E_i is true when the program complies with the prompt, all required correctness checks for input i pass, and both candidate and baseline medians satisfy the timing conditions above (Appendix B.2). The speedup s_i is the ratio of the fastest tested baseline latency to candidate latency. The score weights all $N = 533$ performance cases equally. Failed and unmeasured cases remain in the denominator and contribute zero. Thus fast_0 is the fraction satisfying the source-review, correctness and timing requirements. Default-baseline scores and equal task weighting provide sensitivity analyses (Appendix C.4).

We also report performance restricted to programs that pass every task test. A case contributes only if its program passes all required correctness checks on both performance and correctness-only inputs. The program contributes zero if any required check fails or cannot run. This score retains the same N performance cases. Geometric means use only reported speedups, with case counts stated. A valid speedup requires passing input checks and timings that satisfy the conditions above. Diagnostic flags neither imply incorrect output nor remove the comparison. Prompt compliance additionally determines whether it contributes to the score.

Numerical tolerances. Dot-product tolerances depend on reduction length and the magnitude of the terms being summed. The GCN stack, graph encoder and transformer block use tolerances calibrated against independent implementations, with known limitations (Appendix D.3). Passing a numerical check establishes agreement within the stated tolerance, without proving correctness on every valid input. We compare programs on inputs with valid timings for both. Appendix C.6 defines runtime ratios and the ranking-reversal analysis.

5 EXPERIMENTAL SETUP

We use the requested model identifiers `claude-opus-5` (Opus), `claude-sonnet-5` (Sonnet), `claude-haiku-4-5-20251001` (Haiku) and `gpt-6-astra` (Astra). The exact model snapshots used by the providers were not recorded. Each model produces one program per task across 41 tasks, without GPU access or execution, compilation or profiling feedback. Appendix E gives generation settings and costs.

Prompt and screening. Prompts include the task interface, float64 reference, implementation rules and public descriptions of 328 of the 637 correctness-only cases. Performance-case specifications are not listed (Appendix F.1). Numerical work must be performed by generated kernels implemented using Triton (Tillet et al., 2019) or another permitted toolchain. Each call uses a fresh session, an empty working directory and disabled execution tools. An automated *source screen* checks for GPU kernel code before execution: 41 Opus, 38 Sonnet and 41 Astra programs pass. None of Haiku's 41 programs contains a custom kernel, so all 1,326 cases, including the scientific-matrix study, are rejected before inputs are built. Haiku's results are included in the outcome counts but excluded from speedup analyses. Source review excludes only violations of the prompt's implementation rules (Section 4). All admitted Opus and Astra programs and 35 Sonnet programs comply. Three Sonnet programs delegate numerical work to library operations. AI reviewed Astra's code under author direction without human inspection. The Claude review notes do not identify the reviewer. Both reviews apply the same five rules (Appendix B.5).

Evaluation environment. The three models' programs are timed on NVIDIA L40S GPUs with CUDA 12.8 and PyTorch 2.11.0+cu128, using the same timing procedure and inputs. Compiled-baseline measurements use Python 3.12 and `torch.compile` in default mode with static shapes and partial graph capture. Compilation and warmup precede timing. The candidate, eager baselines and compiled variants are then measured in the same interleaved rotation. The primary comparison includes 652 such measurements across the 23 tasks with eager-composition baselines (Appendix A.5). The same programs handle all sizes and patterns in their tasks. Triangular-solve comparisons use the same baseline adapters for all three models. Appendices B.1, B.2 and E give the software and measurement settings.

A separate study evaluates the same SpMV and SpMM programs on 78 scientific SuiteSparse patterns, selected before measurement by a fixed ordering of content hashes, with one per application domain. Numerical values are generated, and SpMM uses 32 features. These 156 cases per model are reported separately from the synthetic comparison (Appendices A.6 and C.12).

6 Results and Analysis

6.1 Correctness beyond performance inputs

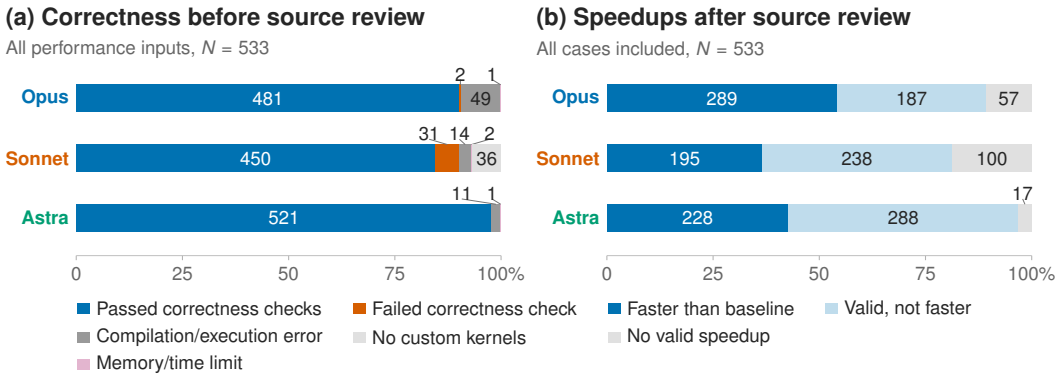


Figure 3: **Correctness checks and performance comparisons.** Each panel includes the primary comparison's 533 performance cases per model. (a) Correctness checks before source review. Programs without a custom kernel were not executed. (b) After source review: valid speedups above one and at most one against the fastest tested baseline, from programs passing source review. Gray means no valid speedup. Labels count cases. All cases remain in the denominator.

Figure 3 separates output checks from speedups after source review on all 533 performance cases per model. Compilation or execution errors and programs lacking custom kernels can prevent output checks. Failed and unmeasured cases remain in the denominator. Of the 164 programs (41 per model), 103 pass every performance-input check, including the output check after timing. Three

nevertheless fail correctness-only inputs: Opus's CSR graph attention network (GAT) layer and Astra's sparse convolution and graph convolutional network (GCN) stack (Table 15).

Astra's convolution passes all ten performance inputs but fails a 256-site, $3 \to 64$ channel case. At one output, six positive products sum to 1.544501, but the program returns 3.085954, outside its 1.09×10^{-6} tolerance. This shape triggers a reduction path absent from the performance inputs (Appendix C.5). The case was described in the generation prompt, but the GAT and GCN failures were not among its examples. All three are public benchmark cases.

6.2 Performance and task correctness

Measure	Population	Opus	Sonnet	Astra
Valid performance comparisons	All 533 cases	476	433	516
fast_1 (per case)	All 533 cases	289 (54.2%)	195 (36.6%)	228 (42.8%)
fast_1 (all task tests pass)	All 533 cases	236 (44.3%)	192 (36.0%)	205 (38.5%)
fast_2 (per case)	All 533 cases	147 (27.6%)	76 (14.3%)	119 (22.3%)
Programs passing all task tests	All 41 programs	32/41	29/41	37/41
GM speedup	395 shared cases	1.37×	0.67×	0.91×
Median speedup	395 shared cases	1.15×	0.86×	0.92×

Table 1: Performance against the fastest tested baseline on the 533 performance cases across 41 tasks. All results require source-review compliance. Percentages retain all 533 cases, including those without a valid speedup. The geometric mean (GM) and median use the same 395 cases with valid comparisons for all three models, without requiring all task tests to pass.

Requiring every task check to pass leaves 236, 192 and 205 faster cases for Opus, Sonnet and Astra (Table 1). Of the 79 comparisons removed from the numerator, 22 belong to the three programs whose failures appear only on dedicated correctness inputs: seven for Opus's GAT layer, ten for Astra's convolution and five for its GCN stack. Another 13 belong to a program already failing performance-input checks. A baseline exhausting GPU memory on one large MTTKRP input prevents an output check and removes 36 comparisons. Compilation or shared-memory errors account for eight more (Table 14). These 44 removals reflect unavailable output checks. The dedicated correctness tests account specifically for the 22 removals from the three newly failing programs.

The aggregate performance fractions, 54.2%, 36.6% and 42.8% for Opus, Sonnet and Astra, measure improvements over the tested implementations across this task population. These improvements can arise from sparse algorithms, fusion of framework operations or avoiding format conversion, so we interpret the aggregate alongside task-level results. All three models outperform the tested baseline on all fifteen sparse-softmax cases. For each model, the fastest baseline records graph capture on thirteen cases, while the eager implementation is faster on the other two. Thus the softmax result includes comparisons against measured compiled reductions. For CSR SpMM, with direct cuSPARSE baselines, Opus, Sonnet and Astra outperform the baseline on 6, 4 and 0 of 33 cases. None does so on the six SpGEMM cases (Figure 6). These rates reflect both generated programs and tested comparators. Where a direct cuSPARSE adapter is fastest, the corresponding counts are 5/70, 4/57 and 0/79 valid comparisons. Where a compiled variant with recorded graph capture is fastest, they are 102/119, 46/111 and 56/128. Each category contains its own input population (Table 10).

On scientific patterns, Opus outperforms the baseline on 8/78 SpMV and 5/78 SpMM inputs, versus 10/30 and 6/33 synthetic inputs. Size, structure and feature width differ. Of the scientific patterns, 61 have fewer nonzeros than every synthetic SpMV performance input. Restricting synthetic SpMM to width 32 and int32 indices leaves Opus faster on 1/11, with size and structure still varying between the populations (Appendices C.12 and C.13).

Sensitivity of the aggregate results. Requiring speedup above $1.10\times$ gives 45.8%, 29.5% and 39.8%, preserving the model ordering. Speedups fall between one and $1.05\times$ on 17, 15 and 10 cases (Table 11). Removing comparisons flagged for the candidate or its selected fastest baseline leaves 274, 189 and 219 faster cases, retaining all 533 cases in the denominator without substituting a slower baseline. Raw medians leave all three fast_1 counts unchanged. Equal task weighting gives 55.5%, 35.2% and 45.0%, also preserving the ordering (Appendices C.4 and C.9). These analyses vary the scoring criteria while holding the recorded measurements fixed.

We also examine dependence on the workloads with limited compiler or hardware support. Excluding the five tasks whose adapters record no graph capture and the float32 2:4 task leaves 484 performance cases across 35 tasks. Opus, Sonnet and Astra outperform the fastest tested baseline on 253 (52.3%), 174 (36.0%) and 201 (41.5%) cases, preserving the model ordering. Failed and unmeasured cases remain in this denominator. The full benchmark remains the primary comparison. This restriction measures the contribution of those six workloads (Appendix C.2).

Baseline choice changes the performance conclusion. On the same primary inputs, 29 Opus, 64 Sonnet and 54 Astra comparisons favor the program against the default baseline but favor the fastest tested baseline (Figure 9). On an embedding-sum input, Opus takes 3.64 ms, the dedicated PyTorch embedding-bag operator takes 17.83 ms, and a CSR matrix-product formulation takes 0.255 ms. The same program has speedup $4.90\times$ against the first and $0.070\times$ against the second (Appendix F.2). Thus baseline choice can overturn an apparent speedup, even between implementations from the same library.

6.3 Sparse structure changes the preferred implementation

We isolate structural variation on CSR SpMM, SDDMM and softmax, with one configuration per task. Dimensions, nonzero count, feature widths and dtypes are fixed across uniform, balanced, banded and skewed inputs. Corresponding inputs share sparse values and dense operands. Each family has 12 reference inputs measured in two independent passes. An ordering requires geometric-mean runtime ratios beyond a 5% margin in the same direction in both passes (Appendix C.14).

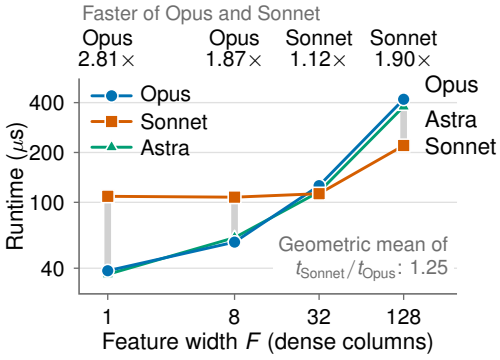


Figure 4: **Feature width changes which CSC SpMM program is faster.** The fixed $65{,}536 \times 65{,}536$ matrix has five diagonals and 327,166 nonzeros. Points show median runtimes minus timing overhead on logarithmic axes, with one timing run per program and input. Lines join measured widths only: the crossing between widths 8 and 32 is not a measured point. Grey bars span Opus–Sonnet runtime gaps, and the row above names the faster of those two programs and its factor. The mean summarizes the four runtime ratios $t_{\text{Sonnet}}/t_{\text{Opus}}$. See Appendix C.8 for construction and schedules.

Input pattern	SpMM $t_{\text{Sonnet}}/t_{\text{Astra}}$	SDDMM $t_{\text{Opus}}/t_{\text{library}}$
Uniform	0.28	1.98
Balanced	0.28	1.98
Banded	0.26	2.07
Skewed	1.45	0.54

Table 2: **Structure changes relative performance.** Geometric mean runtime ratios over both independent timing runs. Values below 1 favor the numerator. Each family has 12 inputs of shape $65{,}536 \times 65{,}536$ with 1,048,576 nonzeros. SpMM has 32 features and SDDMM has 64. Library: the faster passing cuSPARSE or PyTorch sampled addmm on each input.

For SDDMM, Opus takes $1.98\times$ the fastest tested baseline's runtime on uniform inputs, but $0.54\times$ on skewed inputs (Table 2). It is the fastest generated program across all four families. Every uniform reference input favors a library and every skewed input favors Opus in both passes: structure changes whether the library or best generated program is faster. Both declared SDDMM baselines, cuSPARSE and PyTorch sampled addmm, were measured, and the comparison uses the faster passing implementation on each input. Candidate runtimes remain approximately constant across families. Even with a specialized vendor baseline, dimensions and nonzero count alone do not determine the faster implementation: the arrangement of nonzeros changes the comparison.

Structure also reverses the relative performance of one of nine generated program pairs. Sonnet is 3.6–$3.9\times$ faster than Astra on uniform, balanced and banded SpMM inputs, while Astra is $1.45\times$ faster on skewed inputs. Opus remains the fastest generated SpMM program in every family. Three of nine comparisons with the fastest baseline reverse: Opus and Astra on SDDMM, and Sonnet

on softmax. These additional inputs use the declared uncompiled library baselines, so the softmax comparison omits compiled variants (Table 26).

Testing several patterns reveals performance reversals. We compare three ways to choose eight SpMM inputs: two from each of the four families, eight sampled at random from all four, or eight uniform inputs. For each method, we form 100 test sets by resampling the same measured pool of 48 inputs, separate from the reference set. A test set reveals the reversal if it includes both an input where Sonnet is faster and one where Astra is faster, using the 5% margin in both timing passes. This occurs in 97 of 100 sets covering all four families, 84 sampled at random and none using only uniform inputs. All three methods still choose Opus as the fastest generated SpMM program on average (Appendix C.14). Feature width can also change which program is faster, even with the sparse matrix fixed. In Figure 4, Opus is faster at widths 1 and 8, while Sonnet is faster at 32 and 128. The geometric mean of the four runtime ratios favors Opus but hides the advantage of Sonnet at larger widths. Appendix C.8 discusses possible explanations based on the programs' code.

7 LIMITATIONS

Our evaluation covers synthetic inputs and 78 scientific matrix patterns on NVIDIA L40S GPUs with limited data types. The synthetic families provide controlled variation in structural properties, while the scientific study uses application-derived patterns with generated numerical values. These populations differ in their configuration ranges and are not weighted by application frequency (Appendix C.13). The controlled studies show how input structure can change performance comparisons. More tasks and configurations would establish how broadly these effects occur. Other accelerators, lower-precision computations and patterns from learned weights offer further directions for extending the benchmark.

The model study evaluates one program per task and model without execution feedback. Repeated generation and iterative optimization would allow studies of generation reliability and the performance attainable with larger optimization budgets. Comparisons are relative to the tested implementations. Additional specialized baselines and compilation settings could improve this reference set (Appendix A.5). Correctness conclusions apply to the tested inputs and stated numerical tolerances. Independent timing repetitions cover the controlled structural study and selected baseline comparisons. Broader repetition would strengthen conclusions about small performance differences across the full benchmark (Appendices B.2 and C.14).

8 CONCLUSION

SparseKernelBench provides executable sparse tasks with explicit input families, correctness requirements and tested library baselines. The results demonstrate three consequential evaluation choices: sparse structure can change whether a library or a generated program is faster, dedicated correctness inputs can reject programs that pass every performance input, and an alternative library formulation can overturn an apparent speedup. These findings support reporting which inputs a program handles and where it outperforms the tested comparators, alongside aggregate scores. The benchmark supports comparisons of sparse optimizations and code-generation methods under a common execution regime that includes input-dependent preparation on every invocation.

REPRODUCIBILITY STATEMENT

The artifact's `README.md` gives installation instructions and a CPU smoke test. `tools/paper/README.md` explains how to reproduce the tables and figures from recorded results. The supplementary materials include task definitions, baseline adapters, test programs, original generated programs, generation and evaluation scripts, and the analysis plan. The analysis scripts reproduce the performance tables, comparisons on the same inputs and counts of each outcome from full-precision measurements. Review records document whether the programs admitted for timing follow their original prompts. The validation report includes calibration checks, reconstruction of one real matrix without cached data, and an independent baseline audit on the original 38-task core subset, including unsupported cases. Current evaluations check their tested baselines against the reference. The separate audit does not cover every adapter and input in the primary comparison. Full generation transcripts and complete generation-cost records are not retained. Appendices B.5 and F.1 describe the records and analysis. Appendix D.1 describes the validation coverage and its limits.

AI USE STATEMENT

Language models produced the candidate programs evaluated in the case study. Under author direction, AI coding assistants also contributed to the harness, tests, documentation, analysis code and manuscript, and reviewed generated code. The authors are responsible for verifying the implementation, analyses, citations and final manuscript.

REFERENCES

Luca Bertinetto, João F. Henriques, Samuel Albanie, Michela Paganini, and Gül Varol. Preface. In *NeurIPS 2020 Workshop on Pre-registration in Machine Learning*, volume 148 of *Proceedings of Machine Learning Research*, pp. i–i. PMLR, 2021.

Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, 2011. doi:10.1145/2049662.2049663.

Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. An iterative solver benchmark. *Scientific Programming*, 9(4):223–231, 2001. doi:10.1155/2001/527931.

Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2020. doi:10.1109/SC41405.2020.00021.

Yuxuan Han, Meng-Hao Guo, Zhengning Liu, Wenguang Chen, and Shi-Min Hu. Making LLMs optimize multi-scenario CUDA kernels like experts. *arXiv preprint arXiv:2603.07169*, 2026. URL `https://arxiv.org/abs/2603.07169v1`.

Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2020. doi:10.1109/SC41405.2020.00076.

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

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*, 2025. URL `https://arxiv.org/abs/2509.14279v1`.

Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1):2, 2007. doi:10.1145/1217299.1217301.

Shiyang Li, Guangyan Sun, Jinwei Tang, Yanzhi Wang, Mingyi Hong, and Caiwen Ding. SparseDitto: An agentic sparse compilation framework through architecture-aware synthesis on GPUs. *arXiv preprint arXiv:2608.05033*, 2026. URL https://arxiv.org/abs/2608.05033v3.

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

Brian A. Nosek, Charles R. Ebersole, Alexander C. DeHaven, and David T. Mellor. The preregistration revolution. *Proceedings of the National Academy of Sciences*, 115(11):2600–2606, 2018. doi:10.1073/pnas.1708274114.

NVIDIA. cuSPARSE library documentation. https://docs.nvidia.com/cuda/archive/12.8.1/cusparse/, 2025. Version 12.5.8, as bundled in the torch 2.11.0+cu128 wheel; the version is recorded in each result record's environment fingerprint.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47356–47415. PMLR, 2025a.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs Write Efficient GPU Kernels? *arXiv preprint arXiv:2502.10517v1*, 2025b. URL https://arxiv.org/abs/2502.10517v1. February 2025 version.

Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. Compilation of shape operators on sparse arrays. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2), October 2024. doi:10.1145/3689752. URL https://doi.org/10.1145/3689752.

Philippe Tillet, H. T. Kung, and David 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*, pp. 10–19. ACM, 2019. doi:10.1145/3315508.3329973.

Bobby Yan, Alexander J Root, Trevor Gale, David Broman, and Fredrik Kjolstad. Fast autoscheduling for sparse ML frameworks. In *2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 28–43. IEEE, 2026. doi:10.1109/CGO68049.2026.11394842. URL https://doi.org/10.1109/CGO68049.2026.11394842.

Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. SparseTIR: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660–678, 2023. doi:10.1145/3582016.3582047.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max (Xiangjun) Fan, Julian Reed, Azalia Mirhoseini, and Will Su. KernelBench-Verified: Do LLM-generated kernels actually beat PyTorch? *arXiv preprint arXiv:2607.16241*, 2026. doi:10.48550/arXiv.2607.16241.

A Benchmark specification

A.1 What a model receives

The `export-prompt` command generates each task prompt from its task definition and can run on a CPU-only machine. For `csr_spmv`, the prompt includes the following function signature:

```
def csr_spmv(
    *,
    crow_indices: torch.Tensor,
    col_indices: torch.Tensor,
    values: torch.Tensor,
    num_rows: int,
    num_cols: int,
    x: torch.Tensor,
    out: torch.Tensor,      # shape ['M'], float32
                            # row_major_contiguous
    state: object | None = None,   # whatever `prepare` returned, or None
) -> torch.Tensor | None:
```

The prompt also specifies:

- **The operation**, expressed as a sum over the sparse values, including the required behavior for empty rows and explicitly stored zeros.
- **Input requirements**: each operand's dtype, canonical ordering, guaranteed properties and restrictions on modification. The prompt distinguishes prohibited modifications from those the checker can detect.
- **A reference implementation**, using float64 on the CPU to check numerical correctness.
- **Permitted specialization**: dimensions, index and value array lengths, dtypes and device information available as public metadata in the `prepare` hook. Preparation may not use information derived from input coordinates or values, or retain it across calls. The harness checks for such reuse by repeating calls with the same shapes and different coordinates.
- **Public correctness examples**, with case names, shapes, generators, index dtypes and seeds. These overlap the evaluated correctness tests. The performance inputs are not listed (Appendix F.1).
- **Numerical accuracy**: a per-element tolerance formula and its policy identifier. Any summation order is allowed if it meets the stated accuracy requirement. An element with zero exact value and zero condition scale must be exactly zero.
- **Work included in and excluded from timing**. Input generation, host-to-device transfer, structural validation and compilation are excluded. Format conversion, sorting, segment-offset construction and packing are included. A workspace allocated from public metadata in `prepare()` may be reused, while input-dependent contents must be computed within each timed invocation (Appendix B.5).
- **Implementation rules**. Triton, inline CUDA, C++ extensions, `torch.compile` and hand-written PTX are allowed. Calling a sparse library, directly or indirectly, is prohibited because those libraries provide the baselines. Computing the answer densely is also prohibited. A program may decline a configuration with `NotImplementedError`, but that case still remains in the score's denominator.

A.2 Composite layers and model components

These tasks measure forward computations from supplied features, parameters and sparse structures. They cover individual layers and components of larger architectures. Table 3 specifies a composite layer and all six model components. Each returns its final dense feature array. Intermediate arrays need not be materialized. The listed computations, including any required sparse-index construction, occur inside the timed call (Section 4).

Task	Computation and supplied quantities
GCN layer	$Y = \mathrm{ReLU}(A(XW) + b)$ combines projection, sparse aggregation, bias and activation. The CSR aggregation operator A is supplied with its normalization already encoded. The output contains features for each row.
GCN stack	Repeat the GCN layer with separate weights and biases at each layer, reusing the supplied row-normalized CSR operator. ReLU follows every layer, including the last. Return the final node features.
GNN encoder	Sum projections of each node's own features and its mean neighbor features, add a bias and apply ReLU. Follow with single-head graph attention and ReLU, then symmetrically normalized graph convolution with bias and no final activation. The input is a CSR mask with self-loops. Degrees, normalization weights and attention coefficients are computed within the task. Return node features.
Transformer block	Compute query, key and value projections, single-head attention over a supplied CSR mask, and an output projection. Add the residual and apply layer normalization, then apply a two-projection ReLU feed-forward network, another residual and layer normalization. Return token states.
Mixture-of-experts block	Scale token states, apply routed and shared experts, combine their outputs and add the input residual. Each expert has two projections without bias, with ReLU between them. Routing assignments, gate weights, token scales and channel scales are supplied, and their computation is outside the task. Return token states.
Recommender block	Project dense sample features with ReLU, sum weighted item embeddings, multiply the two vectors elementwise and project the result. Item lookup indices and weights are supplied. Return a feature vector per sample without a residual or final activation.
Sparse-convolution block	Find neighbors from supplied 3-D coordinates for two $3 \times 3 \times 3$ convolutions that preserve the active sites. Apply a supplied channel scale and ReLU between convolutions, add a projected shortcut from the original features, then apply ReLU. Return features at the original sites. No normalization statistics are computed.

Table 3: **Computations specified by the composite and model tasks.** All tasks receive their features and model parameters as inputs. Supplied normalization factors are distinguished from normalization computed within the task. These are component forward passes, not complete trained architectures.

A.3 SUITE COVERAGE

The primary comparison covers 41 tasks, 533 performance cases and 637 correctness-only cases (Figure 2). Figure 5 shows the task and generator coverage of the evaluated inputs. We use "sparsity pattern" for the generator-defined grouping recorded as `structure_regime` in the data. The suite-level score gives each performance case equal weight.

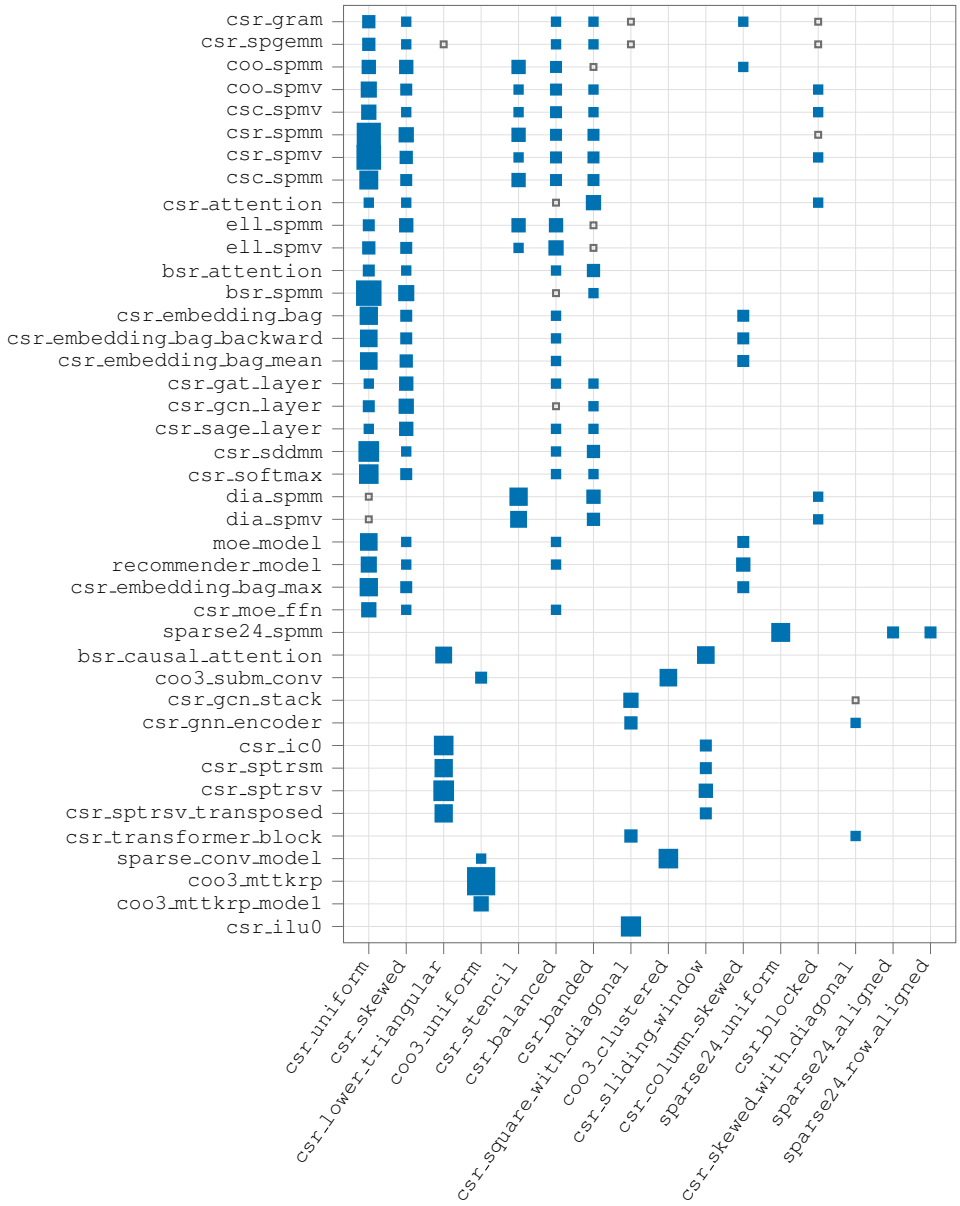


Figure 5: **Task–generator inventory of the study's synthetic inputs.** The 152 pairs cover 41 tasks and 16 structure generators. Blank cells contain no test cases. Filled-square area is proportional to the number of performance cases in the pair (1–30). Open squares mark the 15 pairs containing correctness-only cases and no performance cases. There are 533 performance and 637 correctness-only cases. Tasks are ordered by their number of generators, and generators are ordered by their number of performance cases.

A.4 What the resource tiers contain

Table 4 summarizes the inputs in the primary comparison. The core suite contains 395 small and 22 medium performance cases, together with 637 correctness-only cases in the development tier. The large suite adds 79 medium and 37 large performance cases. The extra-large tier requires at least 80 GiB of device memory and lies outside this evaluation on a 48 GB L40S.

	Development	Small	Medium	Large
Correctness-only cases	637	0	0	0
Performance cases	0	395	101	37
Tasks	41	41	24	16
Structure generators	15	16	9	8
Block shapes	14	5	7	2
Index dtypes	2	2	2	2
Value dtypes	1	1	1	1

Table 4: Input counts in the combined core and large evaluation by resource tier. Counts include every specified input, regardless of its evaluation outcome. Block-shape counts include only tasks with a block parameter.

Generator parameters describe the requested structure. For example, the nonzeros-per-row parameter is a mean for skewed inputs. Two small-tier skewed cases have longest rows of 6,761 and 71,337 entries even though the mean is at most 128. We use statistics of the generated inputs when interpreting performance (Appendix F.2).

Correctness coverage across generators. Performance inputs undergo the same reference comparison as correctness-only inputs before they are timed, and reporting a speedup requires that comparison to pass. There are 24 task–generator pairs with performance cases but no additional correctness-only case. For example, the dedicated correctness cases for triangular vector solve use lower-triangular generators, while its sliding-window inputs appear only in the performance set.

Among the 16 generators, `csr_sliding_window` is the only one used exclusively for performance cases. Its 18 cases span five tasks: 8 for `bsr_causal_attention`, 4 for `csr_sptrsv`, and 2 each for `csr_ic0`, `csr_sptrsm` and `csr_sptrsv_transposed`. Dedicated correctness coverage is also small for `csr_column_skewed`, with 16 performance and three correctness-only cases, and `sparse24_row_aligned`, with two performance and one correctness-only case.

A.5 Baseline implementations and measured work

Table 5 groups the declared adapters by computational family. Direct vendor calls and PyTorch wrappers can share the same backend library. The inventory contains eight direct cuSPARSE adapters, 22 individual framework operators, 17 format conversions followed by an operator, and 32 eager compositions. These categories describe adapter code. A PyTorch operator can itself dispatch to cuSPARSE. Each timed call includes the conversion and preparation needed to compute the output. Creating sparse-tensor wrappers that share existing storage, validating input structure and compiling code remain outside timing. Candidate preparation receives only public metadata and has no access to baseline state derived from input values or coordinates. Each baseline is checked for support on each input, with unsupported configurations retained in the records.

Compiled compositions. For each of the 32 eager compositions, we also evaluate the same callable wrapped in `torch.compile(mode="default", dynamic=False, fullgraph=False)`. The variants supplement the 79 declared adapters. Shapes are fixed within a case. Partial graph capture is allowed, with graph breaks and eager fallback recorded. Baseline preparation first invokes the eager adapter to check support, then compiles it and makes three synchronized warmup calls before timing. Each case attempt starts with empty Inductor and Triton caches. The measured call includes the compiler wrapper's guards and all input-dependent work, but not compilation. Compiled outputs use the same reference check as eager outputs, and compiled variants join the same timing rotation as the candidate and eager baselines. A compiled variant can supply the fastest baseline but never the default baseline. Only default mode is evaluated.

CUDA-graph modes, `max-autotune` and separately hand-fused implementations are outside this comparison.

Six adapters capture no graph in this environment: CSR attention, both GCN layer formulations, the GCN stack, the transformer block and submanifold convolution. Their variants execute eagerly behind the compiler wrapper. Sparse-tensor operations prevent tracing in the graph compositions. For submanifold convolution, Python 3.12 inlines the list comprehensions into a frame that Dynamo skips. All evaluated compiled outputs pass their reference checks. On the largest MTTKRP input, with 134,217,728 nonzeros, both the eager and compiled unfolding formulations fail to build, while the index-add formulations remain available.

Task family	Implementations and configurations	Measured work and support
CSR SpMV/SpMM	cuSPARSE SpMV default; SpMM default and CSR ALG3; PyTorch `mv`, `mm`, `addmm` wrappers	Direct adapters use caller outputs, float32, int32/int64 indices and CUDA. Sparse descriptors are prepared outside timing.
COO/CSC, ELL/DIA products	Native PyTorch products or gather–multiply–reduce; alternative conversion to CSR followed by a product	Format conversion is timed on conversion adapters. Native entry-point support depends on the installed backend.
BSR SpMM	PyTorch BSR `mm`/`addmm`; BSR-to-CSR conversion followed by SpMM	Native PyTorch block products reject nonsquare blocks in the tested environment. The conversion baseline expands BSR indices to CSR inside the timed call and then uses a CSR product.
2:4 SpMM	PyTorch dense, CSR and semi-structured products	Dense reconstruction, CSR index decoding and semi-structured packing are timed. The sparse tensor-core adapter requires float16 or bfloat16 and is unavailable for all evaluated float32 inputs. The fastest measured baseline is the dense product after expansion.
SpGEMM and Gram	PyTorch CSR and COO sparse products	COO paths time conversion, coalescing and CSR output reconstruction. Gram paths also time transpose materialization. Backend support is checked on each input.
SDDMM and sparse softmax	cuSPARSE SDDMM default or PyTorch sampled product; composed CSR or COO softmax	SDDMM uses existing sparse descriptors. CSR-to-COO conversion and composed reductions are timed for softmax.
Embedding bags	Dedicated PyTorch reductions; CSR product formulations; scatter-reduce or index-add alternatives	Constructing sparse indices, computing value products, divisors and per-entry terms, and initializing outputs are timed where required.
Triangular solves; IC(0)/ILU(0)	PyTorch sparse triangular solve; direct cuSPARSE incomplete factorization with and without level scheduling	Solve adapters are CUDA-only. Factorization times value copying and analysis, requires int32 indices and records int64 inputs as unsupported.
Tensor products and convolution	PyTorch gather/index-add or unfolding for MTTKRP; searchsorted/gather/products for submanifold convolution	MTTKRP times unfolding, index casts and per-entry or Khatri–Rao products. Convolution rebuilds its kernel map inside timing.
Composite and model computations	PyTorch sparse/dense compositions; alternative aggregation order, attention layout or expert grouping where declared	Dense products, normalization values, construction of sparse indices and other intermediates are timed. MoE includes token-index expansion, sorting and gathers.

Table 5: Baseline inventory by task family, transcribed from the adapter contracts. The measurement records identify the selected adapters and input types. Input-dependent conversion and intermediate computation are timed. The tested CUDA environment and its observed support limitations are given in Appendices D.1 and E.

A.6 REAL-MATRIX INPUTS

The three real-matrix corpora can be reconstructed from specified versions of public sources. `scientific_v1` records a SuiteSparse collection revision and a checksum for each matrix, while the two activation corpora specify exact model-checkpoint revisions. The harness includes scripts to build and verify these inputs. Appendix D.1 reports a fresh reconstruction of one scientific matrix. The performance study evaluates the sparsity patterns of 78 scientific matrices on CSR SpMV and CSR SpMM, giving 156 inputs per model. It uses generated numerical values and dense operands with the scientific patterns (Appendix C.12). The activation corpora are outside this study.

B EVALUATION PROTOCOL

B.1 CONFIGURATION AND RECONSTRUCTION

The primary comparison evaluates 41 tasks on 533 performance cases and 637 correctness-only cases. The core suite contributes 417 performance cases and all correctness-only cases, while the large suite contributes 116 performance cases. Both use `runtime_sparsity/v1`, task-specific accuracy requirements and `adaptive_median/v5` timing. Numerical tolerances for model tasks are specified in Appendix D.3. Scores use the fixed denominator specified by `strict_fast_p/v4`, including cases with no available baseline. The benchmark's timing rules require candidates and baselines to start from the same inputs and include input-dependent preparation in each timed call. The evaluated suites are reconstructed from `src/sparse_kernelbench/suites/builtin/core.toml` and `src/sparse_kernelbench/suites/builtin/large.toml` at revision `fbe828d`. The selection manifest identifies every case and the measurement used in the results, including its program hash and execution configuration. Each selected row is verified against its raw-store hash, and program source hashes are checked before applying source-review decisions. For 652 model–input comparisons on tasks with eager compositions, the reconstruction also identifies the measurements containing compiled variants, with per-case attempt chains, source hashes and selected rows. Each such measurement times the candidate and all available eager and compiled baselines together. The large-suite population includes all cases up to the `large` resource tier. The suite populations include the three component tasks for mixture-of-experts, recommendation and sparse convolution. Table 7 gives the recorded settings and identifies which checks were enabled.

B.2 TIMING CONFIGURATION AND DIAGNOSTIC CHECKS

Table 7 records the models' generation and measurement settings. Before timing, cache flushing overwrites a byte buffer equal to the device-reported L2-cache size. The recorded multiplier is one. The absolute byte count was not retained. The task specifications and numerical rules agree across models on each input (Appendix D.3).

The five timing measurements. Each implementation is measured in five ways, interleaved within the same run: the scored invocation, a replicate of it, an invocation without a cache flush, and two controls that call the implementation twice and three times per sample. The replicate checks whether the medians agree when the same code is sampled again. The one-, two- and three-call controls estimate a fixed measurement overhead and a cost per invocation. Their second difference tests whether the medians follow this linear model. These controls run without cache flushing because repeated calls share one timed region. Comparing a flushed single call with unflushed repeats would incorrectly attribute the cache-miss cost to measurement overhead. All three controls therefore use warm caches.

Throughput measurements. The supplementary blocks of calls measured with host synchronization target 20 ms, limiting the cost of repeated throughput calls for slow implementations. Block length is chosen from the scored variant in the first round and shared by every variant of that implementation in all rounds (Table 6). The scored metric remains the median CUDA-event time per invocation. With the minimum number of valid samples, warm-up and timing controls require $440 + 72B$ invocations per implementation: 512 when each call takes at least 20 ms, compared

Setting	Recorded configuration and stopping rule
Warm-up	25 invocations per expanded variant, once before the rotation.
Sampling	Three rounds per variant. Each stops at 666 attempted samples, or after both 10 usable samples and $250/3$ ms elapsed wall time. Policy limits are 30–2000 samples. Integer division makes the actual ceiling 1998. Samples from all rounds are pooled before taking the median.
Variant order	All timed baselines then candidate, grouped by scored, replicate, unflushed, doubled and tripled roles. Forward, reverse, forward across rounds.
Cache policy	Fill a reusable byte buffer of one reported L2-cache size with zeros and synchronize before each scored/replicate sample, outside the clocks. Unflushed/doubled/tripled controls do not flush.
Throughput	Three blocks per round, each with $B = \min(50, \max(1, \lfloor 20\,\mathrm{ms}/h \rfloor))$ calls, where h is the first round's host-synchronized median for the scored variant. Every variant and round of an implementation uses the same B. Flush once before each scored block. This separate host-time metric does not determine the score.
Score correction	$t = d - c$, where d is the pooled scored median. To obtain c, we time consecutive CUDA events with no kernel launch between them (200 pairs each round), then take the median of the round medians. Use raw d if $c/d > 0.5$ or the subtraction is nonpositive. Without a usable estimate of c, use both raw medians.
Diagnostic thresholds	Flag relative IQR > 0.25, host/event ratio > 4, replicate deviation > 0.05, fitted offset fraction > 1, or curvature > 0.1. Only host/event disagreement excludes a timing. Other flags, including missing controls and insufficient usable samples, remain diagnostic. A speedup requires finite positive candidate and baseline medians.
Attempt selection	Use the selected last result of each measurement chain. Appendix B.2 gives the manifest and compiled-measurement rules. Independent confirmation passes do not replace the selected measurements.

Table 6: Timing configuration used for all measured models. Appendix B.2 defines the control statistics and scoring conditions.

Setting	Opus	Sonnet	Astra
Client request timeout	Not recorded		
Suite manifest	Evaluated synthetic tasks: 41 tasks, 1170 cases.		
Evaluation limits	3,600 s per case, or 24 h for designated cases and timeout repeats.		
Measurement selection	Last result of each recorded chain. The complete chains, execution limits and selection rules are given in Appendix B.2. Independent confirmation passes are separate.		
Compiled variants	Default mode, static shapes, partial graphs. Compilation and warmup outside timing.		
Cache buffer	One device-reported L2 size, overwritten before scored and replicate samples.		
Timing diagnostics	Relative IQR ≤ 0.25; host/event ≤ 4; replicate deviation ≤ 0.05; offset fraction ≤ 1; curvature ≤ 0.1; instrument share ≤ 0.5. Flags are diagnostic except host/event disagreement. Use the raw median when the overhead share exceeds its limit. See Table 6.		
Same-code controls	Enabled.		
Isolated-baseline check	Disabled.		
Numerical rules	Task-specific tolerances. Model-task rules in Appendix D.3.		
Comparison population	41 tasks, 533 performance cases.		

Table 7: Recorded settings for the three measured models. Scored timing samples overwrite one L2-sized byte buffer before timing. Absolute buffer bytes were not retained.

with 4,040 for fixed 50-call blocks. Adaptive sampling can require more, and these counts exclude correctness checks, preparation and calibration.

Repeated evaluations. Each case runs in a separate process so that a CUDA fault, such as an illegal memory access, cannot invalidate the CUDA context of another case. The standard per-case execution limit is 3600 seconds. Cases assigned the extended limit, and cases repeated after reaching the standard limit, have 24 hours. The recorded measurement chains permit a repeat after an infrastructure failure, a repeat under the extended limit after a standard-limit timeout, and a repeat for a missing candidate or baseline timing comparison, at most once each in that order. Measurements containing compiled variants may also have one repeat for an unavailable compiled comparison. The same rules apply to every model. The selection records retain the complete chains, reasons and execution limits. The last result in each chain counts regardless of its outcome. Independent second passes test the repeatability of these selected measurements.

Definitions of the timing diagnostics. These definitions specify the measurement checks used in the experiments. Let m and r be medians of the pooled raw CUDA-event samples for the scored invocation and its replicate, u the median for a call without cache flushing, and m_2, m_3 the medians for two and three calls. The single-call median used in the fit is

$$s = \begin{cases} u & u > 0, \\ m & u \le 0,\ r \le 0, \\ r & u \le 0,\ r > 0,\ m \le 0, \\ (m + r)/2 & \text{otherwise.} \end{cases}$$

Thus $s = u$ for the recorded runs when the unflushed control has a valid median. For sorted samples $x_0, \ldots, x_{n-1}$, Q_q linearly interpolates between the adjacent samples at position $q(n-1)$.

Statistic	Evaluated expression	Diagnostic threshold or scoring rule
Relative IQR	$(Q_{0.75} - Q_{0.25})/m$ if $m > 0$; otherwise 0	> 0.25 or nonfinite
Replicate deviation	$\lvert r/m - 1\rvert$, defined only for $m, r > 0$	> 0.05 or unavailable/nonfinite
Curvature	$\lvert m_3 - 2m_2 + s\rvert/s$, defined only for $s, m_2, m_3 > 0$	> 0.10 or unavailable/nonfinite
Offset fraction	$k = (m_3 - s)/2$, $o = s - k$, $f = o/(m - o)$; require $m, s, m_3, k > 0$	Signed $f > 1$; $f = +\infty$ if $m - o \le 0$
Host/event ratio	Host-synchronized median divided by m, evaluated only when both are finite and positive	Exclude timing if > 4 or nonfinite
Score correction	Corrected latency $m - e$, with shared empty-event estimate e	Use raw m if $m - e \le 0$ or $e/m > 0.5$

The fitted offset o is separate from the correction e subtracted when computing speedup. Relative spread, replicate disagreement, curvature and offset diagnostics are recorded even when they exceed their thresholds or cannot be computed. Among these diagnostics, only the host/event check affects timing eligibility: a ratio above four or a nonfinite ratio excludes the timing because the synchronized host may have observed work outside the CUDA-event interval. A value equal to a threshold passes that check.

The sampler retains finite, nonnegative durations and discards negative or nonfinite readings. Paired host and CUDA-event readings must both be valid. The record identifies discarded samples, missing controls and insufficient samples. A reported comparison requires finite positive medians for the candidate and at least one baseline that passed its output check. A missing, negative or nonfinite overhead estimate causes the raw medians to be used. Otherwise each side uses $m - e$ unless the estimate exceeds half that side's median or would make it nonpositive. In those cases that side uses m. Nonfinite derived diagnostics are stored as null with the affected field names.

Fitted overhead and the score correction. The repeated-call controls estimate an offset separately for each implementation. The harness subtracts the shared empty-event estimate from that fitted offset to report a diagnostic launch-cost estimate. Speedup uses only the shared empty-event estimate, subtracted from candidate and baseline medians subject to the raw-median fallback above. Appendix B.6 describes the effect of this correction.

B.3 CORRECTNESS CHECKS

Table 8 separates numerical agreement from the other requirements a program must satisfy. The evaluator reuses the prepared program on repeated and transformed inputs, comparing each output with a reference computed for those inputs. These additional calls test whether a program reuses stale outputs or input-dependent state. Only transformations valid for the task are included. A change that leaves a particular input unchanged is recorded as inapplicable.

Check	Applies to	Required behavior
Numerical output	Every checked output	Values satisfy the task's elementwise tolerance against its reference. Nonfinite outputs fail.
Output format	Output on the original input	Shape, dtype, device and layout match the task specification.
Sparse output pattern	Returned CSR matrices	Row pointers and sorted, unique column indices form a valid matrix. Its pattern matches the reference exactly, before comparing values.
Output buffer	Caller-allocated outputs	Write every requested element and return the provided buffer or nothing. A sentinel written before the call detects unwritten elements.
Input preservation and aliasing	Original immutable operands and output arrays	Input digests agree before and after evaluation, and output storage does not overlap input storage.
Repeated calls	Every task	Reusing the prepared program on the same input still gives a correct output. Timed candidates are checked again after timing with the same callable that was measured.
Changed inputs	Applicable task and input combinations	Reflected sparse coordinates and task-specific changes, such as operand scaling, feature permutation or added stored zeros, produce the reference answer for the transformed inputs.
Softmax checks	Sparse row softmax	Shifted logits agree with a recomputed reference, and each nonempty output row sums to one within the row-sum tolerance.

Table 8: Required correctness checks and their scope. Sparse matrix outputs allocate their own storage, so they use pattern checks instead of the caller-buffer check. Transformations apply only when they preserve the task's valid-input requirements.

For transformed inputs, the verdict uses the recomputed reference because rounding can change an ideal scaling or invariance relation. The softmax row-sum tolerance is $4\sqrt{k}u$ for a row with k entries and unit roundoff u.

B.4 ACCURACY REQUIREMENTS AND SUMMATION ORDER

For dot-product tasks the default float32 policy is

$$\mathrm{tol}_i = 2u|\mathrm{ref}_i| + 4\sqrt{k_i}\,u\kappa_i, \qquad \kappa_i = \sum_j |a_j x_j|.$$

Here $u = 2^{-24}$ is float32 unit roundoff, k_i is the number of product terms contributing to output i, and ref_i is its float64 reference value. The scale κ_i sums the absolute values of those terms. The stored label `derived` identifies a tolerance computed from the operation and its operands. The square-root factor models reduction error, in contrast to the worst-case bound $k_i u/(1 - k_i u)$. For example, a valid CSR row with 4,096 distinct columns, values $[1, 2^{-25}, \ldots, 2^{-25}]$, and an all-ones dense operand has exact sum 1.0001220405101776. Sequential float32 accumulation starting with 1 rounds away every remaining term and returns 1. Its error is 1.2204×10^{-4}, whereas the policy allows approximately 1.5380×10^{-5}. This order therefore fails the accuracy requirements despite implementing the specified mathematical reduction.

B.5 REVIEW AGAINST THE GENERATION PROMPTS

The review applies the generation prompts to all 79 Claude and 41 Astra programs admitted by the source screen. Both review records check five prompt requirements: numerical work in custom kernels, no framework computation of the answer, no cached information derived from input coordinates or values, permitted metadata and allocation operations, and workspace use within the timing contract. The records retain decisions and supporting source excerpts. Astra's record describes an author-directed AI source review and explicitly states that no human review was performed. The Claude record for the original tasks does not identify its reviewer. The nine component-task programs from Opus, Sonnet and Astra have separate author-directed AI reviews of their complete sources. These reviews assess compliance from the source code.

A workspace allocated from public metadata in `prepare()` may be reused across calls, provided input-dependent contents are recomputed inside the timed invocation. Framework operations such as scans, sorting, `repeat_interleave` and `bincount` may construct indices and schedules inside the timed call. For example, counting integer indices to build row offsets is allowed, but computing floating-point products in framework code before a custom reduction is not. The prompt allows `torch.compile` and the structural PyTorch operations described above.

The classifications are 41 compliant Opus programs, 35 compliant and three noncompliant Sonnet programs, and 41 compliant Astra programs. The three Sonnet exclusions violate the explicit requirement that numerical work happen in a custom kernel. Its graph encoder delegates dense products, activations and attention-score reductions. Its Gram product computes floating-point products in framework code before a custom reduction. Its transformer delegates dense products and normalization. These programs remain excluded on all inputs, including empty inputs. The review classifies the whole program, so a particular input need not execute the code that violates the rules.

Primary scores use valid measurements from compliant programs, compared with the fastest tested baseline. Their denominator includes all 533 performance cases across the core and large suites. Incorrect outputs, unavailable measurements and excluded programs add nothing to the numerator but remain in the denominator. Table 1 reports these scores using the classifications confirmed by the prompt's author. Direct runtime comparisons use only inputs with valid measurements for both programs in each pair.

B.6 HOW THE TIMING CORRECTION AFFECTS SPEEDUP THRESHOLDS

Let b and c be the baseline and candidate median latencies, and x the shared estimate of measurement overhead. When both sides use the correction and $c > x$,

$$(b - x)/(c - x) - 1 = (b - c)/(c - x),$$

whose sign is the sign of $b - c$ and therefore of $b/c - 1$. When $b = c$, the corrected ratio remains 1.0. Thus subtracting the same correction preserves which implementation is faster. For a threshold above parity,

$$(b - x)/(c - x) - p = (b - pc + x(p - 1))/(c - x),$$

and for $p > 1$ the added $x(p - 1)$ term is positive, so a case can cross p only upward.

The same correction on both sides. Both identities require the same x for the baseline and candidate, with neither side using the raw-median fallback. The scorer uses the candidate's empty-event estimate for both medians. Different corrections can change the ordering: taking $b = 10$, $c = 9$, $x_b = 3$ and $x_c = 1$ gives a raw ratio of $10/9$ above parity and a corrected one of $7/8$ below it.

C COMPLETE RESULTS

C.1 RESULTS FOR THE PRIMARY COMPARISON

Table 1 gives the primary scores over the core and large suites. Figure 6 groups all 41 tasks by computational level, showing valid speedups and cases on which programs outperform the fastest tested baseline. Each denominator retains all performance cases for its task. For CSR triangular-vector, transposed-vector and triangular-matrix solves, valid measurements include bandwidths of 1 and 256 as well as wider bands and scattered inputs.

In the transposed solve, Astra's non-diagonal branch fails Triton compilation on 24 cases because its loads combine incompatible cache and volatile modifiers. These failures occur before the program produces an output. Its 2 successful cases use the diagonal branch. Sonnet's program contains no custom kernel and fails the source screen on all 26 cases. Opus produces valid measurements on all 11 performance inputs. Table 9 separates incorrect outputs, execution failures and unavailable timing comparisons across the full suite.

One Sonnet transformer-block case with $1{,}048{,}576$ rows times out under the 24-hour execution limit. It remains in the performance denominator and has no speedup. The limit covers output checks, baseline work and timing controls as well as candidate execution.

C.2 Baseline implementations

Table 10 separates the primary results by the kind of implementation used as the fastest tested baseline on each input. Categories describe the adapter's entry point on each input. A single PyTorch sparse operator can dispatch internally to cuSPARSE, and a task's fastest baseline can belong to different categories on different inputs. The compiler categories use the selected baseline's recorded graph count on the same input. Graph capture can cover only part of an operation, and the counters can omit graphs reused from an earlier build in the same process.

The baseline set includes compiled variants of the 32 eager compositions (Appendix A.5). Six adapters execute eagerly because the wrapper captures no graph. No compiled baseline output fails its numerical reference check. On the largest MTTKRP input, both versions of the unfolding adapter fail to build, while the compiled index-add implementation runs and supplies a comparison.

Task-population sensitivity. As a post hoc sensitivity analysis, we remove the five tasks associated with the six adapters that record no graph capture: CSR attention (8 performance cases), GCN layer (8), GCN stack (5), transformer block (4) and submanifold convolution (10). We also remove float32 2:4 SpMM (14), whose comparisons use dense/CSR implementations. Together, these six tasks contribute 49 cases per model, of which Opus, Sonnet and Astra outperform the fastest tested baseline on 36, 21 and 27. Subtracting these cases from the full comparison gives 253/484, 174/484 and 201/484, respectively. All cases in the retained 35 tasks stay in the denominator, including failures and unavailable comparisons. The valid-comparison counts are 430, 401 and 467. The measurements, baseline selection and source-review requirements are unchanged, so the comparison measures the contribution of these six tasks to the aggregate.

C.3 Sensitivity to performance margins and timing flags

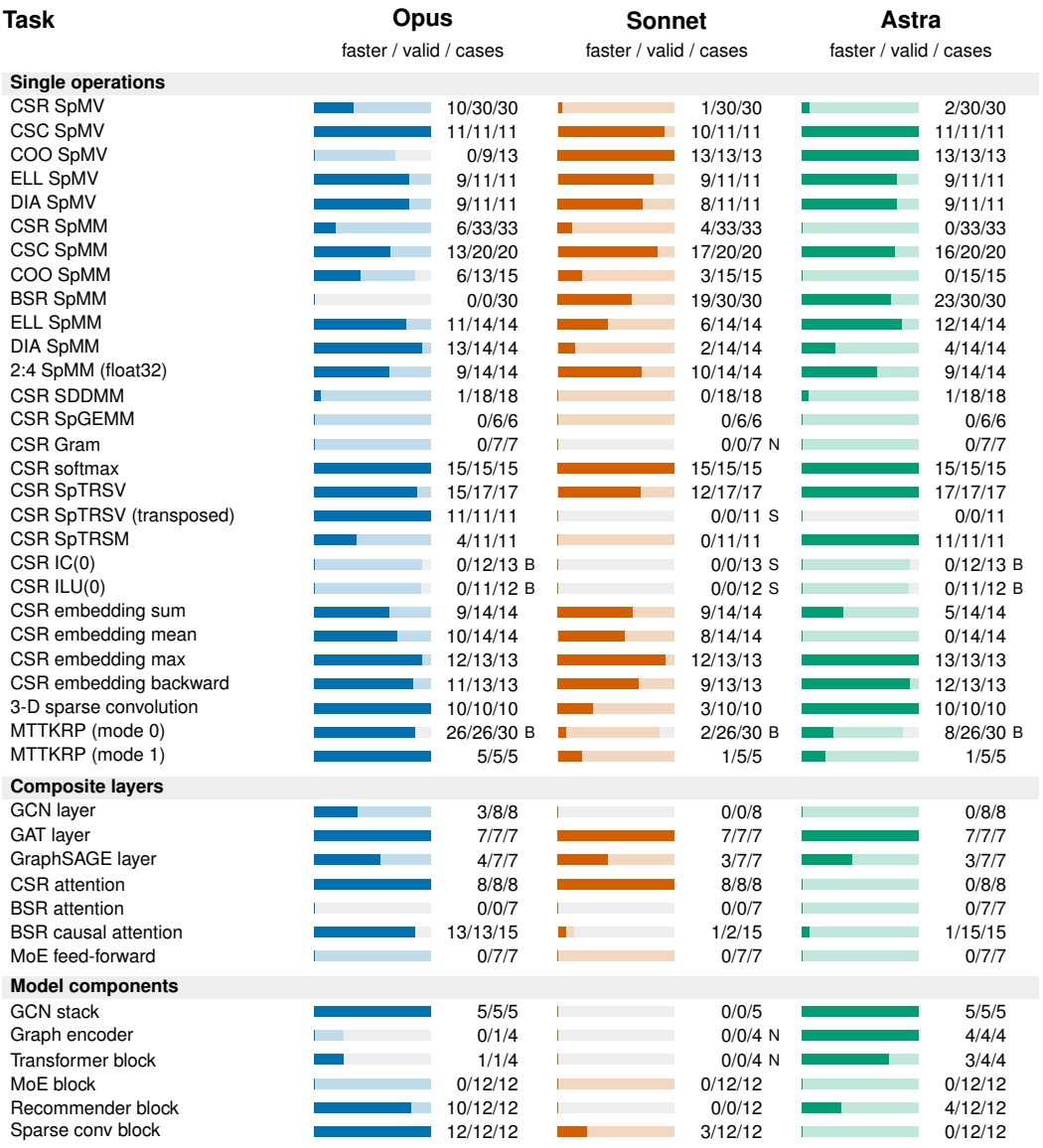
The primary score uses a strict speedup threshold of one. Raising it to 1.05 or 1.10 tests dependence on small improvements, using unrounded measurements. The flag diagnostic removes a credited comparison when the candidate or its already-selected fastest baseline has any recorded timing quality flag without substituting a slower baseline. Shared timing calibration has no refused measurements in this population. Each sensitivity analysis retains every performance case in the denominator. Its result is reported separately from the primary score in Table 1. Recorded flags concern within-run agreement, variation and timing overhead (Appendix B.2).

C.4 Weighting and correctness across all task inputs

The primary score weights performance cases equally, so tasks with more cases contribute more. To assess this weighting, define a score that weights each task in the evaluated set $\mathcal{T}$ equally:

$$\mathrm{MacroFast}_p = \frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \frac{\sum_{i \in t} 1[E_i \wedge s_i > p]}{N_t},$$

where N_t is the number of performance cases for task t and E_i is the eligibility indicator in Section 4. All 41 tasks receive equal weight, including those without a valid speedup. Against the fastest tested baseline at $p = 1$, Opus, Sonnet and Astra obtain 55.5%, 35.2% and 45.0%, respectively, using the primary score's measurements and eligibility rules. This removes differences due to the number of cases per task. Operation families with more task variants still receive more weight.



Task	Opus faster / valid / cases	Sonnet faster / valid / cases	Astra faster / valid / cases
Single operations			
CSR SpMV	10/30/30	1/30/30	2/30/30
CSC SpMV	11/11/11	10/11/11	11/11/11
COO SpMV	0/9/13	13/13/13	13/13/13
ELL SpMV	9/11/11	9/11/11	9/11/11
DIA SpMV	9/11/11	8/11/11	9/11/11
CSR SpMM	6/33/33	4/33/33	0/33/33
CSC SpMM	13/20/20	17/20/20	16/20/20
COO SpMM	6/13/15	3/15/15	0/15/15
BSR SpMM	0/0/30	19/30/30	23/30/30
ELL SpMM	11/14/14	6/14/14	12/14/14
DIA SpMM	13/14/14	2/14/14	4/14/14
2:4 SpMM (float32)	9/14/14	10/14/14	9/14/14
CSR SDDMM	1/18/18	0/18/18	1/18/18
CSR SpGEMM	0/6/6	0/6/6	0/6/6
CSR Gram	0/7/7	0/0/7 N	0/7/7
CSR softmax	15/15/15	15/15/15	15/15/15
CSR SpTRSV	15/17/17	12/17/17	17/17/17
CSR SpTRSV (transposed)	11/11/11	0/0/11 S	0/0/11
CSR SpTRSM	4/11/11	0/11/11	11/11/11
CSR IC(0)	0/12/13 B	0/0/13 S	0/12/13 B
CSR ILU(0)	0/11/12 B	0/0/12 S	0/11/12 B
CSR embedding sum	9/14/14	9/14/14	5/14/14
CSR embedding mean	10/14/14	8/14/14	0/14/14
CSR embedding max	12/13/13	12/13/13	13/13/13
CSR embedding backward	11/13/13	9/13/13	12/13/13
3-D sparse convolution	10/10/10	3/10/10	10/10/10
MTTKRP (mode 0)	26/26/30 B	2/26/30 B	8/26/30 B
MTTKRP (mode 1)	5/5/5	1/5/5	1/5/5
Composite layers			
GCN layer	3/8/8	0/0/8	0/8/8
GAT layer	7/7/7	7/7/7	7/7/7
GraphSAGE layer	4/7/7	3/7/7	3/7/7
CSR attention	8/8/8	8/8/8	0/8/8
BSR attention	0/0/7	0/0/7	0/7/7
BSR causal attention	13/13/15	1/2/15	1/15/15
MoE feed-forward	0/7/7	0/7/7	0/7/7
Model components			
GCN stack	5/5/5	0/0/5	5/5/5
Graph encoder	0/1/4	0/0/4 N	4/4/4
Transformer block	1/1/4	0/0/4 N	3/4/4
MoE block	0/12/12	0/12/12	0/12/12
Recommender block	10/12/12	0/0/12	4/12/12
Sparse conv block	12/12/12	3/12/12	0/12/12

Dark: faster than baseline; pale: valid speedups; gray: no valid speedup.
N: violates rules; S: rejected by source screen; B: baseline unavailable on at least one case.

Figure 6: **Valid measurements and speedups across the 41 evaluated tasks.** Each bar uses all performance cases for that task as its denominator. Pale bars show valid speedups from programs passing source review, and dark bars show cases faster than the fastest tested baseline. Numbers give faster cases, valid speedups and total cases. Gray includes programs excluded by source review, numerical and execution failures, and unavailable or invalid measurements. Letters identify violations of the implementation rules, source-screen rejections and unavailable baselines. The float32 2:4 inputs compare against dense/CSR implementations and do not exercise sparse tensor-core acceleration.

failure stage	evaluation outcome	Opus	Sonnet	Astra	Haiku
correctness	incorrect	14	67	2	0
correctness	runtime_failure	10	15	24	0
correctness	unsupported_configuration	23	0	0	0
execute	timeout	0	1	0	0
import	compile_failure	50	0	0	0
import	policy_violation	0	79	0	1170
none	baseline_unavailable	5	3	5	0
prepare	compile_failure	4	0	0	0
timing	out_of_memory	1	1	1	0

Table 9: Failure stage and evaluation outcome for each selected case without a correct result, including both kinds of case. Excluding programs after source review does not change the evaluation outcomes. A failure at the correctness stage need not produce an answer or a numerical verdict.

Fastest baseline	Opus 5	Sonnet 5	Astra
cuSPARSE call	5/70	4/57	0/79
One PyTorch operator	89/144	72/147	78/147
Format conversion + operator	31/60	49/65	52/73
Eager composition	60/81	24/53	42/89
`torch.compile`: graphs recorded	102/119	46/111	56/128
`torch.compile`: no graphs recorded	2/2	—	—
All scored cases	289/476	195/433	228/516

Table 10: **Performance by the fastest baseline's implementation category.** Each cell is the number of cases faster than the baseline divided by the number of valid comparisons. A dash denotes no valid comparisons. Only cases from programs that pass source review enter either count. The tested baselines include compiled versions. A case enters the category of its fastest measured baseline. These are model-specific populations, not comparisons on identical inputs across models. Categories describe an adapter's entry point. A PyTorch operator can itself invoke a vendor library. Compiler rows are separated by whether the selected variant recorded graph capture during its build. Capture can be partial and does not imply complete fusion. Counts can omit graphs reused from an earlier build in the same process.

The correctness-only column in Table 13 counts inputs, with 637 per model. Pass means the output satisfies the numerical and input/output requirements. All failures in this column are outputs outside the allowed numerical tolerance: 12 for Opus, 36 for Sonnet and 2 for Astra. Missing means that no output was checked: 38 inputs for Opus, 44 for Sonnet and 13 for Astra. Compilation or execution errors or programs that did not define a custom kernel prevented these comparisons. Of Sonnet's 44 such cases, 43 belong to programs that did not define a custom kernel. All 637 Haiku correctness-only cases were rejected for the same reason, before execution.

The remaining columns count programs, with 41 per model. A program passes all task tests only if every required output check passes on every input. An observed failure or an input without a correctness result prevents this classification. Inapplicable transformation checks may be skipped. Table 15 separates failures from cases whose outputs could not be checked. The numerical rules are specified in Appendix D.3.

Counting speedups only when programs pass every task check and comply with the implementation rules gives 236, 192 and 205 cases on which programs outperform the fastest tested baseline for Opus, Sonnet and Astra, respectively, over all 533 performance cases. The corresponding rates are 44.3%, 36.0% and 38.5%. This rule requires the program to pass every task input, whereas the case-level score considers each performance case separately. All performance cases remain in the denominator, including those from programs with failed checks or inputs whose outputs could not be checked.

Comparison	Opus	Sonnet	Astra
fast_1	289/533	195/533	228/533
$\text{fast}_{1.05}$	272/533	180/533	218/533
$\text{fast}_{1.10}$	244/533	157/533	212/533
fast_1 after removing flagged comparisons	274/533	189/533	219/533
Cases with $1 < s \leq 1.05$	17	15	10

Table 11: **Sensitivity to performance margins and timing flags.** All fractions retain the same 533 performance cases, including failed and unmeasured cases. The timing-flag diagnostic removes comparisons with a flag on the candidate or its selected fastest baseline. It does not choose a slower baseline. The first row reports the primary score. Performance margins are not confidence intervals.

		Opus		Sonnet		Astra	
Workload level	Cases	$s > 1$ / valid	$s > 1$ when all tests pass	$s > 1$ / valid	$s > 1$ when all tests pass	$s > 1$ / valid	$s > 1$ when all tests pass
Single operations	425	226 / 383	194	173 / 378	171	201 / 408	183
Composite layers	59	35 / 50	15	19 / 31	18	11 / 59	11
Model components	49	28 / 43	27	3 / 24	3	16 / 49	11

Table 12: Performance by workload level, before and after requiring all task tests to pass. Each model's first column gives cases with $s > 1$ / valid speedups. The second counts such cases only for programs passing every numerical and contract check on the task's performance and correctness-only inputs. Rows partition the 533 performance cases of the primary comparison. Valid speedups require compliance with the implementation rules, an accepted numerical output and valid program and baseline timings, as in Table 1. A value $s > 1$ means the program is faster than the fastest tested baseline. The Cases column includes all performance inputs. Each model's columns count only those satisfying the stated conditions.

	Correctness-only	All task inputs			All checks passed
Model	pass / fail / missing	Pass	Fail	Missing	and compliant
Opus	587 / 12 / 38	32	2	7	32
Sonnet	557 / 36 / 44	31	4	6	29
Astra	622 / 2 / 13	37	2	2	37
Haiku	0 / 0 / 637	0	0	41	0

Table 13: Correctness across every input of a task. For correctness-only inputs, pass means all tests passed, fail means numerical error exceeded tolerance, and missing means a required output check has no result. The correctness-only column covers 637 dedicated test inputs per model, and the remaining columns count 41 programs per model. Pass requires every required check to pass on every input, including the output check after timing on performance inputs. Fail takes precedence over missing results. The final column additionally requires source-review compliance. Numerical rules are specified in Appendices C.4 and D.3.

	Primary	All tests	Cases excluded	
Model	$s > 1$	passed: $s > 1$	Failed task checks	Output not checked
Opus	289	236	20	33
Sonnet	195	192	0	3
Astra	228	205	15	8

Table 14: Requiring every task test to pass reduces the number of cases with $s > 1$, where the program is faster than the fastest tested baseline. All columns refer to the same 533 performance cases. The last two columns separate programs with a failed numerical or contract check on any task input from those with unavailable output checks. Compilation or execution errors and GPU memory exhaustion can prevent these checks.

Model	Performance-input checks	Correctness-only inputs Pass	Fail	Missing
Opus	All checks passed	32	1	0
Opus	Failed check	0	1	0
Opus	Missing result	4	0	3
Sonnet	All checks passed	31	0	0
Sonnet	Failed check	0	4	0
Sonnet	Missing result	2	0	4
Astra	All checks passed	37	2	0
Astra	Failed check	0	0	0
Astra	Missing result	1	0	1
Haiku	All checks passed	0	0	0
Haiku	Failed check	0	0	0
Haiku	Missing result	0	0	41

Table 15: Performance-input checks and dedicated correctness tests, before source review. Each model has 41 programs. Passing performance-input checks includes passing the output check after timing. Missing means at least one input has no correctness result and no check failed, due to compilation or execution errors or programs that did not define a custom kernel. Dedicated tests expose three failures after all performance-input checks pass: Opus's CSR GAT layer and Astra's 3-D sparse convolution and GCN stack.

The correctness-only inputs reveal failures in the Opus GAT-layer program and Astra's sparse-convolution and GCN-stack programs, which pass the checks on every performance input. The causal-attention example in Appendix C.11 shows failures on additional input shapes, although the program already fails 2 performance inputs. The rule depends on the numerical checks: changing acceptance on a correctness-only input can change which programs contribute speedups without changing any timed output.

C.5 Correctness-only inputs expose untested execution paths

Astra's sparse-convolution program passes the output checks on all 10 performance inputs but fails `coo3_subm_conv/contract/narrow_input_wide_output`. The input has 256 lexicographically sorted active sites in a $12 \times 12 \times 12$ grid, 32-bit indices, 3 input channels and 64 output channels. Features and weights are finite float32 values sampled from a normal distribution. The initial reference comparison rejects all $256 \cdot 64 = 16{,}384$ output elements. Output-shape, buffer, immutability and aliasing checks pass.

For a concrete example, consider output channel 60 at site $(0, 7, 4)$, using zero-based indexing. Only that site and $(0, 6, 5)$ are active neighbors. Their six feature–weight products are approximately

$$\underbrace{0.0114851 + 0.1194453 + 0.9386888}_{(0,6,5)} + \underbrace{0.1805692 + 0.0746919 + 0.2196205}_{(0,7,4)}.$$

Using the exact float32 inputs, this sum is 1.54450072547. The recorded GPU output is 3.08595442772 and its tolerance is 1.0861132×10^{-6}. Every product is positive, so this sum involves no cancellation. A CPU reconstruction rounds the six products to float32 and enumerates all $6! = 720$ sequential addition orders. Their results lie between 1.54450058937 and 1.54450082779, all within the tolerance.

Source inspection identifies the path reached by this shape:

```
if sites == 1 or (input_channels <= 8 and output_channels <= 16):
    fused_lookup_and_reduction(...)
elif input_channels >= 16 and output_channels >= 16:
    build_neighbor_map(...); tiled_dot_products(...)
else:
    build_neighbor_map(...); explicit_product_reduction(...)
```

On 9 performance inputs, the program selects tiled dot products, while the remaining $1 \to 1$ case selects the fused path. The failing $3 \to 64$ case reaches the explicit product reduction, which no

performance input exercises. The same path passes the $32 \to 8$ and $64 \to 3$ correctness inputs. Its written reduction expresses the required sum, but the cause of the failing GPU result remains unresolved because intermediate values were not captured. The artifact includes the operands and a CPU reconstruction of the reference sum.

C.6 Comparisons on the same inputs and sensitivity

We compare programs on the same inputs, requiring valid speedups and compliance with the generation prompt for both programs. Table 16 summarizes their runtime ratios on these shared inputs, including tasks that do not have enough patterns for the reversal analysis. We determine the shared inputs and eligible tasks separately for each model pair. Tables 17 and 18 report how their valid measurements overlap and which patterns reverse the ordering. Figure 8 shows all eligible tasks.

For programs A and B, the ratio $r_i = t_{B,i}/t_{A,i}$ uses candidate times directly. We use the ordered pairs (A, B) = (Opus, Sonnet), (Astra, Opus) and (Astra, Sonnet) throughout these comparisons. Values above one favor A. We take geometric means over each pattern and all shared cases of a task. A task is eligible for ranking-reversal analysis if two patterns each have at least two shared cases. A reversal occurs when its overall mean and an eligible pattern's mean lie on opposite sides of one. A reversal exceeds margin m when both lie outside $[1/(1+m), 1+m]$. The margin specifies a minimum performance difference, not statistical significance.

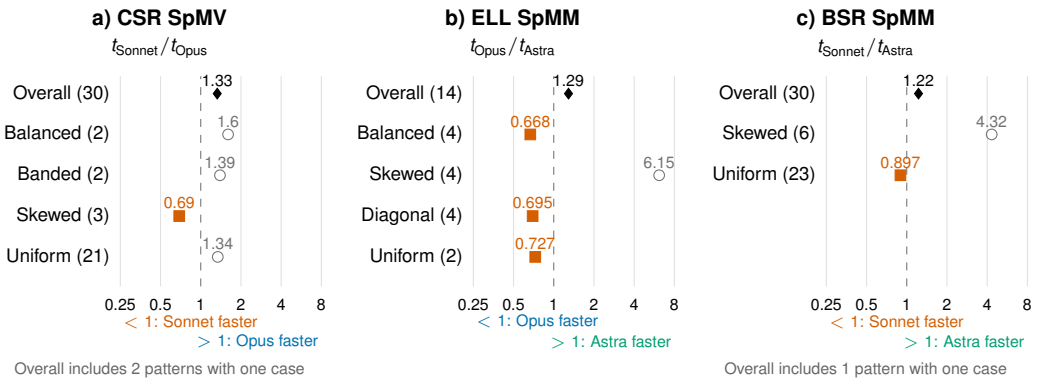


Figure 7: **A program can be faster overall but slower on a particular pattern.** These three examples have runtime differences above 10% both overall and on a pattern with the opposite ordering. Table 19 summarizes the benefit of choosing a program separately for each pattern. Diamonds show geometric means of runtime ratios over shared cases, where t denotes candidate runtime. Squares mark patterns whose ordering is opposite to the task mean, and circles mark the remaining patterns. Parentheses give shared-case counts. Task means include all shared cases. Plotted patterns have at least two. Axes share a log scale. Figure 8 reports all eligible comparisons.

Pattern-level comparisons. Figure 7 shows runtime-order reversals above 10%: 4 of 21 tasks for Opus–Sonnet, 2 of 25 for Astra–Opus and 5 of 22 for Astra–Sonnet. Figure 8 reports all comparisons. These pattern groups vary jointly in structure, dimensions, density and feature width.

Choosing separately by pattern gives geometric-mean speedups of $1.013\times$, $1.018\times$ and $1.044\times$ over one choice per task for Opus–Sonnet, Astra–Opus and Astra–Sonnet. This comparison selects and assesses programs on the same inputs and excludes the cost of a runtime dispatcher (Table 19).

Programs (A, B)	Ratio	Cases	GM	Median	Faster: A / B / ties
Opus, Sonnet	$t_{\text{Sonnet}}/t_{\text{Opus}}$	395	2.031	1.312	294 / 100 / 1
Astra, Opus	$t_{\text{Opus}}/t_{\text{Astra}}$	465	0.627	0.857	171 / 293 / 1
Astra, Sonnet	$t_{\text{Sonnet}}/t_{\text{Astra}}$	433	1.312	1.109	243 / 190 / 0

Table 16: Candidate runtimes on the same inputs (41 tasks). Ratios divide B's runtime by A's, so values above one mean A is faster. GM and median summarize these individual ratios. The last column counts cases where each program is faster or their recorded runtimes are equal. The implementation rules match Table 1. Both programs must have valid speedups, although the runtime ratio itself contains no baseline term. The inputs with valid speedups for both programs are determined separately for each pair.

programs (first, second)	total	both	first only	second only	neither
Opus, Sonnet	533	395	81	38	19
Astra, Opus	533	465	51	11	6
Astra, Sonnet	533	433	83	0	17

Table 17: Cases with valid speedups for both programs, only one or neither, after source review on all 533 performance cases of the evaluated core and large suites (41 tasks). Both means both programs have a valid comparison with the fastest tested baseline. Neither means neither has a valid speedup and does not identify the cause.

Task	N	Task ratio	Reversing pattern	n	Pattern ratio
Opus, Sonnet ($t_{\text{Sonnet}}/t_{\text{Opus}}$)					
CSC SpMM	20	0.792	Stencil	4	1.254
CSR embedding backward	13	1.209	Column-skewed	2	0.566
CSR embedding max	13	1.058	Skewed rows	2	0.995
CSR SpMV	30	1.329	Skewed rows	3	0.690
DIA SpMV	11	1.116	Stencil	7	0.897
Astra, Opus ($t_{\text{Opus}}/t_{\text{Astra}}$)					
CSC SpMV	11	0.965	Balanced rows	2	1.032
CSR embedding max	13	1.138	Column-skewed	2	0.917
			Uniform	9	0.992
CSR embedding mean	14	0.637	Skewed rows	3	1.715
DIA SpMV	11	0.858	Banded	3	1.023
ELL SpMM	14	1.289	Balanced rows	4	0.668
			Stencil	4	0.695
			Uniform	2	0.727
ELL SpMV	11	1.295	Balanced rows	5	0.971
Astra, Sonnet ($t_{\text{Sonnet}}/t_{\text{Astra}}$)					
BSR SpMM	30	1.224	Uniform	23	0.897
COO SpMM	15	1.212	Stencil	4	0.368
CSC SpMM	20	0.921	Stencil	4	1.310
			Uniform	10	1.049
CSR embedding backward	13	1.427	Column-skewed	2	0.964
CSR embedding mean	14	0.681	Skewed rows	3	1.717
CSR SpMV	30	1.125	Skewed rows	3	0.615
CSR triangular solve	17	2.108	Sliding window	4	0.654
DIA SpMM	14	1.348	Banded	4	0.970
DIA SpMV	11	0.958	Banded	3	1.216

Table 18: Sparsity patterns that reverse the overall runtime ordering for a task in Figure 8. Group headings name programs (A, B) and give the ratio t_B/t_A. Values above one favor A. N counts all shared task cases, and n counts the pattern's shared cases. Both ratios are geometric means and lie on opposite sides of one in every row. The input population and source-review decisions match the figure.



Figure 8: **Relative performance can reverse across sparsity patterns.** Panels compare program runtimes on identical inputs. The shared cases differ by model pair. The model names give programs (A, B), and ratios are t_B/t_A. Values above one favor A. Marks show geometric means of these runtime ratios. Axis labels name the faster model on each side of parity. Orange squares mark patterns whose runtime ordering is opposite to the task's overall ordering. Outlined squares have a runtime difference below 1% for the task or pattern. Gray connectors span the shown patterns, not confidence intervals. Each task includes all shared cases. Eligible tasks have at least two patterns with $n \geq 2$. Columns give shared cases per task. Dashes mean no eligible comparison. Shown tasks cover 317/395 (Opus–Sonnet), 362/465 (Astra–Opus), 353/433 (Astra–Sonnet) cases shared by the compared programs. Table 18 names the reversing patterns. All included programs pass source review.

Cost of selecting one implementation per task. For each eligible task and pair of programs, we first choose the program with the lower geometric-mean runtime across all shared cases. We then choose separately for each sparsity pattern using the same criterion, including patterns represented by a single case. We evaluate both choices on those same inputs and report the geometric mean of the runtime with one choice per task divided by the runtime with one choice per pattern. This measures the possible benefit of selecting by pattern on the observed inputs. Table 19 combines results across cases and reports the examples from Figure 7.

Pair	Cases (tasks)	Overall speedup	Task with largest speedup	Task cases	Task speedup
Opus–Sonnet	317 (21)	1.013×	CSR embedding-bag backward	13	1.091×
Astra–Opus	362 (25)	1.018×	ELL SpMM	14	1.303×
Astra–Sonnet	353 (22)	1.044×	COO SpMM	15	1.305×

Table 19: Speedup from choosing an implementation by sparsity pattern on recorded inputs. For each model pair, we compare choosing one program per task with choosing separately for each sparsity pattern. Each choice minimizes the geometric-mean corrected runtime across its inputs. We report the geometric mean of the per-case runtime ratio: one choice per task divided by one choice per pattern. Cases receive equal weight, and input sets differ by pair. Included tasks have at least two patterns with at least two shared cases each. All patterns of these tasks contribute, including those represented by one case. Both choices are selected and evaluated on the same recorded inputs.

C.7 Implementation choices on six tasks

Astra uses inline CUDA in six programs: Gram product, SpGEMM, triangular matrix solve, triangular vector solve, incomplete Cholesky and incomplete LU, with 22 CUDA kernel definitions in total. Neither Opus nor Sonnet uses inline CUDA. The Claude sources for Gram product, SpGEMM and triangular matrix solve use framework operations for sorting, index expansion, scans or dependency scheduling. On the other three tasks, the Claude sources include host implementations, with conditional fallback paths in the GPU-capable programs.

Astra implements the corresponding structural work in custom kernels, including histogram, transpose construction and scans for Gram product, and output-layout construction for SpGEMM. These source-level comparisons show different approaches to structural work across the same tasks. The transposed triangular-vector solve is a separate case: Astra uses Triton, and its non-diagonal branch fails during compilation on 24 of 26 inputs before producing an answer, as described above.

C.8 CSC SpMM strategies in the feature-width sweep

The width sweep fixes a $65{,}536 \times 65{,}536$ five-diagonal matrix with offsets $0, \pm 1, \pm 256$. Entries are retained whenever the flattened column index is in range, giving $65{,}536 + 2(65{,}535) + 2(65{,}280) = 327{,}166$ nonzeros. This construction includes 510 connections across flattened row boundaries that a nonperiodic 256×256 five-point spatial grid would omit. Sparse pointers, indices and values are identical across widths 1, 8, 32 and 128. Source review classifies both Claude programs as compliant with the implementation rules. The following pseudocode summarizes the two programs.

Opus: tiles of nonzeros	Sonnet: one program per column
For each entry in a tile:	Load the column's dense row $B[j,:]$.
Find column j by binary search in the CSC pointers.	For each entry in column j:
Load its row i, value a and $B[j,:]$.	Load its row i and value a.
Atomically add $aB[j,:]$ to $Y[i,:]$.	Atomically add $aB[j,:]$ to $Y[i,:]$.

Opus uses 64, 64, 32 and 8 nonzeros per tile at the four widths, respectively, producing 5,112, 5,112, 10,224 and 40,896 Triton program instances. Sonnet launches one Triton program instance per column, for 65,536 instances at each width. A CPU reconstruction assigns all 327,166 entries to the same columns under both schedules. Source-level dense-element loads are $327{,}166F$ for Opus and $65{,}536F$ for Sonnet, where F is feature width. Both express $327{,}166F$ atomic additions. These counts exclude initialization kernels. They suggest a tradeoff between work partitioning and dense-row reuse, whose effect on the crossover would require profiling.

C.9 Measurement diagnostics

Timing correction on a fixed population. Table 20 holds prompt compliance, measurement availability and the selected fastest baseline fixed on the primary population. It compares the reported

runtime ratio with the ratio of the same raw medians. Any difference therefore comes from the timing correction, with the included cases and baseline choices unchanged. The correction fraction divides the shared empty-event estimate by the smaller raw runtime. No pair in this analysis uses the raw-median fallback. The same cases contribute to fast_1 with and without the correction. The correction adds 2 fast_2 successes for each model. Geometric means use 476 Opus cases, 433 Sonnet cases and 516 Astra cases, so they describe different input sets.

		Cases with $s > 2$		Geometric mean		Overhead / smaller raw time		
Model	Eligible	Raw	Corrected	Raw	Corrected	Median	95th	Max.
Opus	476	145	147	1.23	1.24	1.7%	14.2%	29.9%
Sonnet	433	74	76	0.751	0.751	1.7%	13.1%	27.3%
Astra	516	117	119	0.819	0.82	1.7%	13.7%	27.0%

Table 20: Effect of subtracting the measured timing overhead. Raw and corrected ratios use exactly the same eligible measurements and the same selected fastest baseline for each case. Threshold counts retain all 533 performance cases. Geometric means use only the stated eligible cases. Subtraction adds 2 cases with $s > 2$ for each model. The same cases satisfy $s > 1$ (289, 195, 228 cases). The last three columns divide the measured overhead by the smaller of the two raw medians. Quantiles use linear interpolation.

Host conditions and baseline timing comparisons. Each evaluation times baselines alongside its candidate. To compare baseline timings across evaluations, we match input fingerprints and baseline identifiers on cases with reported speedups from compliant programs. Both baseline measurements must pass their numerical comparison and the comparison between host and CUDA-event timings. Other timing flags remain diagnostic. Times use the recorded overhead correction and fallback rules from Appendix B.2. Table 21 summarizes these measurements of the same baseline on the same input. Each case–baseline pair receives equal weight.

Time ratio	Cases	Baseline pairs	Geometric mean	5th–95th percentiles
Opus / Sonnet	395	1099	1.001	0.978–1.023
Astra / Opus	465	1305	0.997	0.969–1.019
Astra / Sonnet	433	1193	0.999	0.974–1.018

Table 21: Times of the same baseline on the same input across model evaluations, restricted to compliant programs with reported speedups on the same input. Baseline outputs pass their numerical checks, and timing diagnostics remain recorded. A ratio above one means the baseline was slower during the evaluation named first. Percentiles describe the observed distribution, not uncertainty intervals. A case can contribute several baseline pairs.

Opus/Sonnet: median 1.000, $1075/1099$ pairs within a symmetric 5% margin; Astra/Opus: median 1.000, $1275/1305$ pairs within a symmetric 5% margin; Astra/Sonnet: median 1.000, $1175/1193$ pairs within a symmetric 5% margin.

C.10 A WARM-UP BUFFER SMALLER THAN THE REQUESTED COMPUTATION

Opus's COO SpMV program limits its dummy warm-up arrays to approximately 2^{22} entries but passes the input's full nonzero count to the kernel. For larger inputs, the kernel can read beyond these buffers. Preparation fails with an illegal memory access on 4 of its 13 performance cases. The other 9 performance cases and all 13 correctness-only cases pass. The table gives the measured input properties for every performance case, with the `csr_` prefix omitted from generator names.

nonzeros	longest row	rows	generator	index	outcome
1,048,576	18	262,144	uniform	int32	correct
1,048,576	37	65,536	uniform	int32	correct
1,309,694	5	262,144	stencil	int32	correct
3,199,984	36	199,999	uniform	int32	correct
4,193,935	128	262,144	blocked	int32	correct
4,194,304	41	262,144	uniform	int32	correct
4,194,304	16	262,144	balanced	int32	correct
4,194,304	16	262,144	balanced	int64	correct
4,194,304	16	262,144	banded	int32	correct
8,388,608	6,761	262,144	skewed	int32	fails
8,388,608	71,337	262,144	skewed	int32	fails
16,777,216	103	262,144	uniform	int32	fails
16,777,216	38	1,048,576	uniform	int32	fails

All 4 cases at $2^{22} = 4{,}194{,}304$ nonzeros pass, while the 4 failures have 2^{23} or 2^{24} nonzeros. Passing and failing inputs include overlapping row counts and row lengths, consistent with the total nonzero count controlling this allocation error. The warm-up code has a separate branch for counts not divisible by 16. All 4 failing inputs exercise the branch for multiples of 16.

The warm-up call triggers compilation before evaluating the input, so this defect prevents correctness and timing comparisons on larger cases. Testing only inputs at or below 2^{22} nonzeros would miss this defect. On all 9 completed performance inputs, the program is slower than the baseline.

C.11 Correctness and speed depend on the selected code path

Missing timings can reflect incorrect outputs, so the cases with timings need not represent the full suite. Reweighting them to the suite's intended proportions would assume otherwise. We therefore report missing measurements and correctness-only outcomes explicitly.

Opus's block-sparse causal-attention program selects between two implementations based on the number of rows per block, B_r. At $B_r \leq 64$ it uses a gather implementation, while above 64 it uses a tiled implementation. The following table gives their correctness outcomes over the task's 30 cases.

path	condition	correct	incorrect
tiled	$B_r > 64$	13	0
gather	$B_r \leq 64$	4	13

Of the task's 15 performance cases, 13 have 128×128 blocks and take the tiled path. All 13 pass correctness checks and yield valid speedups. The remaining 2 performance cases use $B_r \leq 64$ and fail the output check before timing. The errors are large: 192,953 of 262,144 elements exceed tolerance in one case, with a worst error-to-tolerance ratio of 2.39×10^6. The other has 393,006 of 524,288 elements outside tolerance and a worst ratio of 9.82×10^5.

Every reported speedup for this task therefore measures the successful path on one block shape, while the same program is wrong on 13 of the task's 30 cases. A single speedup per operation would conceal those failures.

Across the complete case sets, 11 of 15 correctness-only inputs fail, compared with 2 of 15 performance inputs. Conditioning on the gather branch gives failure rates of 11 of 15 correctness-only inputs and 2 of 2 performance inputs. The other 13 performance cases take the tiled branch and pass. The correctness-only set therefore broadens the observed failures to additional block row sizes, while the performance set already contains 2 failures of this program.

The gather implementation passes 4 cases. Outcomes differ for 2 cases with the same shape: both have a 2×2 block shape, 8×8 extents and a 4×4 block grid, and use different query, key and value scales. The case with peaked probabilities passes, while the nearly uniform case fails. Possible explanations include normalization, value or indexing errors that interact with attention weights. The precise faulty operation remains unresolved. One 1×1 grid case passes and another fails, and the remaining 2 passing gather cases have block row sizes 1 and 3.

C.12 PERFORMANCE ON SCIENTIFIC MATRIX PATTERNS

We evaluate the same generated programs on CSR SpMV and CSR SpMM using the 78 scientific sparsity patterns described in Appendix A.6. We report these inputs separately from the 533-case synthetic comparison. Each operation contributes 78 performance cases per model, with generated numerical values and dense operands. Tables 22 and 23 retain every input in the performance fractions and compare geometric means only on the cases with valid measurements for all three models. The timing and source-review rules are the same as in the primary comparison.

All 78 inputs yield a performance comparison for Opus, Sonnet and Astra in both operations. Opus, Sonnet and Astra outperform the fastest tested baseline on 8, 0 and 1 SpMV inputs, respectively. The corresponding SpMM counts are 5, 0 and 0. No program exceeds a $2\times$ speedup on either operation, and all 6 geometric-mean speedups are below 1.

Measure	Opus	Sonnet	Astra
fast_0	78 (100.0%)	78 (100.0%)	78 (100.0%)
fast_1	8 (10.3%)	0 (0.0%)	1 (1.3%)
fast_2	0 (0.0%)	0 (0.0%)	0 (0.0%)
GM speedup (same cases)	0.4507	0.5099	0.3659
Median speedup (same cases)	0.7993	0.7329	0.5471

Table 22: CSR SpMV on the real-matrix study. Performance is relative to the fastest tested baseline. All fractions use 78 performance cases per model, including failed and unmeasured cases. Only comparisons from programs that pass source review contribute to these statistics. fast_0 counts valid comparisons, and fast_1 and fast_2 count speedups above one and two. The geometric mean (GM) and median use the same 78 cases with valid comparisons for all three models.

Measure	Opus	Sonnet	Astra
fast_0	78 (100.0%)	78 (100.0%)	78 (100.0%)
fast_1	5 (6.4%)	0 (0.0%)	0 (0.0%)
fast_2	0 (0.0%)	0 (0.0%)	0 (0.0%)
GM speedup (same cases)	0.5647	0.3193	0.1732
Median speedup (same cases)	0.5792	0.4571	0.1748

Table 23: CSR SpMM on the real-matrix study. Performance is relative to the fastest tested baseline. All fractions use 78 performance cases per model, including failed and unmeasured cases. Only comparisons from programs that pass source review contribute to these statistics. fast_0 counts valid comparisons, and fast_1 and fast_2 count speedups above one and two. The geometric mean (GM) and median use the same 78 cases with valid comparisons for all three models.

C.13 COVERAGE OF SYNTHETIC AND SCIENTIFIC MATRIX INPUTS

We compare the actual input statistics recorded for the synthetic and scientific CSR SpMV and SpMM measurements in Table 24. The scientific subset chooses one matrix per available SuiteSparse application category, using a fixed hash order among matrices with at most four million nonzeros. This gives each category equal representation, independent of its frequency in applications. The study uses the source matrices' sparsity patterns with generated numerical values.

The structural summaries use the materialized sparse operands. Repeated synthetic cases with the same dimensions, generator parameters and structure seed count once, even when their index dtype, dense feature width or numerical values differ. Scientific patterns are identified by their content hashes. The accompanying analysis records quartiles as well as ranges, row-length variation, maximum row length, empty-row fraction and mean row span divided by the number of columns. Row span measures the column-index range within a row, giving a partial measure of locality without describing the full neighborhood structure.

The input populations have substantial differences. Of the 78 scientific patterns, 61 have fewer nonzeros than the smallest synthetic SpMV performance input, and 40 fall below the synthetic

SpMM minimum. Scientific mean row lengths range from 3.11 to 2,202.20, compared with 4–64 for synthetic SpMV and 4.99–32 for synthetic SpMM. Thus, even these simple properties show regions absent from the synthetic performance cases. The reported ranges describe each property separately, while input configurations combine them.

The lower panel counts cases faster than the baseline within each operation, retaining failed and unmeasured cases in each denominator. It also restricts synthetic cases to the scientific study's dtypes and, for SpMM, feature width 32. The same program source is used in both populations for every model and operation. Dimensions, density and structure remain different after matching dtype and width. The comparison measures performance on these two input populations, with their joint differences intact.

Actual input property	Synthetic SpMV	Synthetic SpMM	Scientific
Rows	65,536–4,194,304	16,384–1,048,576	480–1,000,005
Nonzeros	1,048,576–268,435,456	262,144–33,554,432	12,224–3,897,557
Mean nonzeros/row	4–64	4.99–32	3.11–2,202
Row-length CV	0–11.6	0–6.72	0.0138–55.1
Maximum nonzeros/row	5–71,339	5–64,418	4–120,065
Empty-row fraction	0–0.368	0–0.1	0–0.242
Mean row span / columns	0.00189–0.969	0.0078–0.882	0.000787–0.984

Operation	Input population	Opus	Sonnet	Astra
CSR SpMV	All synthetic cases	10/30	1/30	2/30
CSR SpMV	Synthetic, int32	9/22	1/22	1/22
CSR SpMV	Scientific patterns	8/78	0/78	1/78
CSR SpMM	All synthetic cases	6/33	4/33	0/33
CSR SpMM	Synthetic, int32, width 32	1/11	0/11	0/11
CSR SpMM	Scientific patterns	5/78	0/78	0/78

Table 24: **Measured synthetic and scientific CSR input populations.** Upper panel: ranges of materialized input statistics. CV is the standard deviation of row length divided by its mean, and row span is averaged over nonempty rows. The scientific patterns are the same for both operations. Lower panel: fractions of cases with a valid speedup strictly above one against the fastest tested baseline. Source compliance is required. Failed and unmeasured cases remain in each denominator. All values are float32. Scientific cases use int32 indices and SpMM width 32. Dimensions and structure remain different after matching dtype and width. The ranges summarize each structural statistic separately.

C.14 Structural coverage at fixed dimensions and nonzero count

Design and inputs. We compare the unchanged Opus, Sonnet and Astra programs on one configuration each of CSR SpMM, sampled dense–dense matrix multiplication (SDDMM) and sparse softmax. SpMM and SDDMM use $65{,}536 \times 65{,}536$ matrices with 1,048,576 nonzeros and feature widths 32 and 64, respectively. Softmax uses a $262{,}144 \times 262{,}144$ matrix with 4,194,304 nonzeros. All use float32 values and int32 indices. These configurations compare uniform, balanced, banded and skewed coordinates at the same dimensions, actual nonzero count and feature width. For each realization, sparse values and dense operands are byte-identical across the four families. The study was designed after inspecting earlier evaluation results. Its final scope and test selections were fixed before inspecting the new runtime measurements. The library baselines in this controlled study are the declared uncompiled implementations. Compiled variants are not measured on these additional inputs. Performance orderings require both timing passes to satisfy the timing-quality checks.

For each configuration, the selectable pool contains 12 inputs per family and the reference set contains another 12 per family. Twelve further inputs keep one uniform coordinate pattern and redraw only the numerical values. The 324 inputs are materialized in the measuring environment. Their recorded fingerprints verify that programs receive the same inputs and that no reference coordinate pattern occurs in the selectable pool or the fixed-pattern inputs. Both timing passes use the same materialized inputs, with different input orders in separate jobs.

The families change measured structural properties. Across the three configurations, the skewed family's median row-length coefficient of variation (standard deviation divided by mean) is 5.8–6.4, compared with 0.25 for uniform inputs. Its median longest row contains 9,284–17,320 nonzeros, compared with 35–37. Skewed inputs have 10–22% empty rows, whereas uniform inputs have none. Balanced and banded inputs give every row the same length but differ in column locality. The experiment varies these structural properties together while holding dimensions, nonzero count, width and values fixed.

Confirmed orderings. For each program pair and family, we compute the geometric mean of runtime ratios over reference inputs with valid times for both programs in both passes, using the same input subset in each pass. At least six inputs are required. An ordering is confirmed only when both pass-specific ratios lie outside $[1/1.05, 1.05]$ in the same direction. A reversal requires two families with confirmed opposite orderings. All other comparisons are unclassified. Comparisons against a baseline use the same rule, with either a named baseline or the fastest of the baselines measured in this study on each input. The latter can change identity.

Across families, 1 of the 9 program pairs reverses: Sonnet versus Astra on SpMM. Reversals also occur in 3 of 9 comparisons against the fastest baseline measured in this study: Opus and Astra on SDDMM, and Sonnet on softmax. Of 27 comparisons with individual baseline implementations, 5 reverse. On SDDMM, Opus and Astra each reverse against both cuSPARSE and PyTorch sampled addmm. Sonnet also reverses against the composed PyTorch softmax baseline. All reversals involve the skewed family. No family contains reference inputs with confirmed orderings in both directions for a program pair. The SpMM selectable pool does contain 1 skewed input favoring Sonnet, 10 favoring Astra and 1 within the margin. This explains why skewed-only selections can detect both orderings even though none of that family's reference inputs clearly favors Sonnet.

Selecting the same number of tests. For budgets of 2, 4 and 8 inputs per configuration, we compare redrawing values at fixed coordinates, drawing from each family alone, drawing at random from the four-family pool and spreading tests as evenly as possible over the families. When the budget is smaller than four, the last method chooses which families to cover at random. Each method uses 100 fixed selections without replacement within a selection. A selection detects a reversal when its inputs include confirmed per-input orderings in both directions. Percentages summarize selection variability within this fixed measured pool.

At 8 tests, spreading inputs over families detects the Sonnet/Astra SpMM reversal in 97% of selections, random selection in 84% and uniform-only testing in none. These percentages concern that one reversing pair. No selection reports a reversal for the other 8 program pairs. For Sonnet versus the composed PyTorch softmax baseline, skewed-only testing detects both orderings in 100% of 8-test selections, compared with 83% when tests are spread over families. Some skewed inputs make Sonnet faster than that baseline, while others make it slower. By contrast, skewed-only testing never detects either of Opus's SDDMM reversals against named baselines, because all its selectable skewed inputs favor Opus.

The comparison holds test counts equal. Mean evaluation cost at 8 tests, summed over the three programs and including repeated evaluations, is 338 seconds for tests spread over families, 339 seconds for random selection, 359 seconds for uniform-only testing and 351 seconds for value-only testing. Evaluation time therefore varies by about 6% at the same test count.

Measurement checks and limits. Each of the 9 programs passes output checks on all 108 inputs of its task in both passes. In the second pass, 36 Astra softmax inputs lack an accepted candidate timing and are omitted wherever both passes are required, with the remaining reference counts shown in Table 25. For the 984 program–input pairs with valid times in both passes, including the embedding-bag replication in Appendix F.2, the 5th–95th percentile range of the second/first runtime ratio is 0.993–1.006. Corresponding same-baseline ratios across program jobs within a pass have a range of 0.986–1.019. Dividing each program's time by a baseline measured in its own job preserves all 9 program-pair statuses, including the SpMM reversal.

Finding reversals and choosing one program are different objectives. Every method chooses the reference set's fastest program whenever it makes a choice. Some selections cannot choose because they lack timings shared by all eligible programs. Uniform-only testing also reproduces all

Task	Runtime ratio	Uniform	Balanced	Banded	Skewed
SpMM	Opus/Sonnet	0.82/0.82	0.82/0.82	0.83/0.83	0.22/0.22
SpMM	Opus/Astra	0.23/0.23	0.23/0.23	0.21/0.21	0.32/0.32
SpMM	Sonnet/Astra	0.28/0.28	0.28/0.28	0.26/0.26	1.45/1.45
SDDMM	Opus/Sonnet	0.16/0.16	0.16/0.16	0.15/0.15	0.16/0.16
SDDMM	Opus/Astra	0.85/0.85	0.85/0.85	0.83/0.83	0.86/0.86
SDDMM	Sonnet/Astra	5.35/5.35	5.35/5.37	5.60/5.59	5.36/5.36
Softmax	Opus/Sonnet	0.96/0.96	0.99/0.99	0.99/0.99	0.59/0.59
Softmax	Opus/Astra	1.50/1.50	1.49/1.50	1.49/1.50	1.82/1.82
Softmax	Sonnet/Astra	1.57/1.57	1.50/1.51	1.50/1.51	3.11/3.11

Table 25: **All program pairs in the controlled structural study.** Cells give geometric mean runtime ratios in pass 1/pass 2 on shared reference inputs whose timings pass the quality checks in both passes. There are 12 shared reference inputs per family except Softmax Opus/Astra: 6/6/9/12; Softmax Sonnet/Astra: 6/6/9/12 (in column order). Ratios are the first program's runtime divided by the second's. Values below 1 favor the first. Each family has 12 reference inputs. Dimensions, nonzero count and feature width are fixed within each task. An ordering requires ratios below $1/1.05$ in both passes or above 1.05 in both.

Task	Program	Uniform	Balanced	Banded	Skewed
(a) Every program versus its fastest tested baseline					
SpMM	Opus	1.19/1.37	1.44/1.46	1.35/1.35	2.28/2.34
SpMM	Sonnet	1.53/1.44	1.71/1.70	1.63/1.62	10.63/10.46
SpMM	Astra	5.89/5.90	6.26/6.25	6.33/6.33	7.23/7.22
SDDMM	Opus	1.98/1.98	1.98/1.98	2.07/2.07	0.54/0.54
SDDMM	Sonnet	12.40/12.41	12.39/12.40	14.04/14.02	3.36/3.36
SDDMM	Astra	2.32/2.32	2.31/2.31	2.51/2.51	0.63/0.63
Softmax	Opus	0.30/0.30	0.30/0.30	0.30/0.30	0.86/0.86
Softmax	Sonnet	0.31/0.31	0.30/0.30	0.30/0.30	1.48/1.48
Softmax	Astra	0.20/0.20	0.20/0.20	0.20/0.20	0.47/0.47
(b) Comparisons with named baselines that reverse					
SDDMM	Opus/cuSPARSE	1.98/1.98	1.98/1.98	2.07/2.07	0.53/0.53
SDDMM	Opus/sampled addmm	1.38/1.38	1.38/1.38	1.42/1.42	0.52/0.52
SDDMM	Astra/cuSPARSE	2.32/2.32	2.31/2.31	2.51/2.51	0.62/0.62
SDDMM	Astra/sampled addmm	1.63/1.63	1.63/1.62	1.72/1.72	0.61/0.61
Softmax	Sonnet/composed softmax	0.31/0.31	0.30/0.30	0.30/0.30	1.48/1.48

Table 26: **Sparse structure changes whether a program outperforms a library baseline.** Cells are geometric mean program/baseline runtime ratios in pass 1/pass 2. There are 12 shared reference inputs per family except Softmax Astra: 6/6/9/12 (in column order). Below 1 favors the program. These additional inputs measure the declared library baselines without compiled variants. Panel (a) uses the fastest correct baseline with valid timing on each input, so its identity may vary across inputs. Panel (b) lists comparisons with named baselines whose runtime ordering changes across families. The other named comparisons retain one ordering. The PyTorch sampled-addmm and composed-softmax names denote the tested framework entry points.

9 aggregate reference orderings. Across these three configurations and four families, broader coverage exposes performance variation hidden by the aggregate while leaving the selected program unchanged. All programs pass the output checks, so the study assesses performance variation only. The artifact provides the experiment plans, input statistics and per-method results.

Test selection	Both orderings (%)	Same families as reference (%)
Fixed coordinates, new values	0 / 0 / 0	0 / 0 / 0
Uniform only	0 / 0 / 0	0 / 0 / 0
Balanced only	0 / 0 / 0	0 / 0 / 0
Banded only	0 / 0 / 0	0 / 0 / 0
Skewed only	14 / 29 / 63	0 / 0 / 0
Spread across families	40 / 82 / 97	40 / 82 / 97
Random from diverse pool	33 / 62 / 84	33 / 62 / 84

Table 27: **Detecting the SpMM Sonnet–Astra reversal.** Entries are percentages of 100 selections at 2 / 4 / 8 tests. The first column of rates requires selected inputs to exhibit both confirmed orderings. The second additionally requires each ordering to occur in a family with that ordering in the reference set. Skewed-only selections can show both directions within their own family, but cannot reproduce the reference reversal between families. The remaining 8 program pairs show no detected reversal. These are resamplings of one fixed pool, not 100 independent GPU experiments.

D VALIDATION

D.1 VALIDATION OF BASELINES AND NUMERICAL CHECKS

The baseline-validation procedure checks each declared case–adapter pair against the reference, recording unsupported inputs, incorrect outputs and other failures. We report an independent audit of the original core subset below. The primary evaluation also checks each tested baseline on its input.

An independent baseline-validation run on the original 38-task core subset accounts for all 1,956 expected pairs over the 988 core v40 cases. Of these, 1,819 pass their reference comparison and 137 have no numerical comparison. Three baseline adapters could not run any of their declared cases in this environment, accounting for 87 pairs. The other 50 refusals depend on the input: 28 involve empty index arrays, 14 involve BSR block shapes, and eight involve 64-bit indices passed to adapters that require 32-bit indices. Of these 50 pairs, 42 are for correctness-only cases and eight for performance cases. The empty-index refusals arise during sparse-tensor construction in the tested PyTorch version. The inputs remain part of the benchmark even though library support can change with versions.

In this validation run, 383 of 387 performance cases have at least one adapter that passes the reference comparison, while four have none. Two are `bsr_spmm` cases with 8×32 blocks and feature widths 32 and 128. The two PyTorch adapters reject nonsquare blocks, and the third adapter does not support CUDA. The other two are `csr_ic0` and `csr_ilu0` cases with 64-bit indices, while both declared adapters require 32-bit indices. The validation report lists their full case identifiers and refusal reasons. Thus, this audit validates a baseline on $383/387 = 99.0\%$ of its performance inputs. Table 9 reports baseline availability for the full 533-case primary evaluation.

Every completed numerical comparison passes. Comparing expected and recorded case–adapter identifiers finds no missing, duplicate or unexpected pair. Both adapters for triangular vector solves pass all 52 declared comparisons. These results identify the inputs supported by the tested baselines on the recorded L40S environment under the recorded numerical rules. Table 28 separately tests the numerical tolerances: all 472 held-out outputs pass, and all 59 unit-error probes are rejected in each of the CPU and L40S audits. These checks use the base inputs before metamorphic transformations.

Validation also includes reconstructing the SuiteSparse matrix `AG-Monien/grid2_dual` without cached files. We verify both the downloaded MatrixMarket source and its canonical sparsity pattern against the stored checksums before constructing an input for a supported task. This reconstruction checks the download and input-construction procedure. The report records the hardware, commands and outcomes.

D.2 TESTS OF THE EVALUATION SOFTWARE

Regression tests exercise the evaluator with 115 programs containing deliberate errors, such as incorrect indexing, changing an input or reusing an earlier answer. The evaluator catches the intended error in 114 programs on at least one test case, counting only rejections for the intended reason. One program escapes detection: it changes an input and restores it exactly before returning, so comparing the input before and after execution reveals no change. Separate tests use PyTorch implementations to check acceptance of correct answers and verify the timing procedure.

D.3 CALIBRATING NUMERICAL TOLERANCES FOR MODEL TASKS

For the GCN stack, GNN encoder and transformer block, we calibrate numerical tolerances using independent implementations on the reference operands. Their errors relative to the float64 reference determine the tolerance, independently of the generated program's output. The other three model components—mixture of experts, recommendation and sparse convolution—use operation-specific forward-error tolerances from their task definitions.

Tolerance. An output element i is accepted when its absolute error is at most

$$\mathrm{tol}_i = \begin{cases} 0 & \text{if every calibration implementation matches the reference exactly,} \\ 2u\,|\mathrm{ref}_i| + 8Cu\kappa_i & \text{otherwise,} \end{cases}$$

where u is unit roundoff and κ_i is a float64 scale for output element i, computed by propagating absolute input, weight and bias magnitudes through the layers, leaving the scale unchanged at ReLU and using the reference attention weights and layer-normalization divisors where applicable. The calibration factor C is the maximum observed error divided by $u\kappa_i$ over the case's elements and all calibration implementations. The factor eight is a fixed margin. There is no absolute floor: an element whose reference value and condition scale are both zero must be exactly zero. When every calibration implementation matches the reference exactly, the check requires exact agreement. Nonfinite or invalid calibration prevents a numerical verdict.

Each task uses four calibration implementations that change the association of operations, tiling or accumulation order. The GNN encoder and GCN stack use transform-first, aggregate-first, split-halves and thirds-reversed. The transformer block uses two-pass, one-pass-folded, banded-halves and thirds-reversed. Calibration requires at least two implementations and rejects float64 outputs: promotion to reference precision would make the estimated rounding error nearly zero.

Validation. We test the tolerance on implementations excluded from calibration, checking that valid outputs pass and deliberate errors fail. Eight CPU implementations must remain within the tolerance: a row loop, reversed neighbor order, a pairwise tree, compensated summation, a transposed GEMM, row tiling, column tiling and the bias moved inside the aggregation. On the held-out unit-test grid, the maximum ratio of error to tolerance lies between 0.05 and 0.6. Four implementations computing a different function must exceed the tolerance by more than a hundredfold: leaky ReLU in place of ReLU, the last layer's bias dropped, the self-loop dropped from the adjacency and the bias added after the activation instead of before. Two further errors are task-specific: the encoder's numerical check rejects a readout normalized by row degree alone, and the transformer block's check rejects the unbiased variance estimator.

Table 28 extends these checks to all 59 calibrated core inputs at their declared sizes and seeds, using eight held-out implementations per case. All 472 CPU outputs pass. Adding one to the first reference output element is rejected on every input. A separate L40S audit uses eight CUDA float32 implementations per input, with TF32 disabled. All 472 outputs pass and all 59 unit-error probes are rejected. In both audits, six correctness-only inputs produce exact agreement among calibration implementations, and all 48 corresponding held-out outputs also agree exactly with the reference.

Calibration estimates rounding error from a finite set of implementations, whereas the tolerances in Appendix B.4 follow from the operation's arithmetic. Its validation is limited to the tested implementations and base inputs before metamorphic transformations. Generated outputs receive their own numerical checks against the reference.

Environment	Outputs accepted	Unit-error probes rejected
Apple CPU	472 / 472	59 / 59
NVIDIA L40S	472 / 472	59 / 59

Table 28: **CPU and GPU validation of numerical tolerances.** Each audit covers all 59 calibration inputs before metamorphic transformations, at their declared sizes and seeds, with eight implementations excluded from calibration for each case. A unit-error probe adds one to the first reference output element. In each audit, the six cases where calibration implementations agree exactly also have exact agreement among the held-out implementations. CUDA outputs use float32 with TF32 disabled. These checks use library implementations. Generated programs are evaluated separately.

D.4 Source screening

The source screen parses Python code for custom-kernel markers and prohibited operations before execution. A source-policy violation prevents execution and records a policy violation for every case of that program. Compilation, import and execution errors are recorded separately as evaluation failures.

Programs rejected by the screen. Across the three Claude models, the screen rejects all 41 programs from Haiku, 3 from Sonnet and none from Opus. The three rejected Sonnet programs are `csr_ic0` at 27 cases, `csr_ilu0` at 26 and `csr_sptrsv_transposed` at 26, giving 79 Sonnet cases recorded under that outcome. All three are rejected because the screen finds no recognized custom-kernel marker. Haiku's programs are rejected on all 1,326 cases, including the separate scientific-matrix study.

The three rejected programs use two approaches. `csr_sptrsv_transposed` solves by level-set wavefront in tensor primitives—`bincount`, `nonzero`, `cumsum`, `repeat_interleave`—with one Python loop, over levels rather than rows, and no host transfer. `csr_ic0` and `csr_ilu0` copy the CSR arrays to the host and factorize using nested Python loops, operating on NumPy arrays and Python lists, respectively. These programs express dependencies between rows through framework operations or host loops, without implementing a custom kernel.

E Generation details

Table 29 maps the short names to the literal model identifiers recorded in the generation summaries. The original program snapshots are dated September 14, 2026 for Opus, Sonnet and Haiku, and September 16, 2026 for Astra. Individual request timestamps and provider-resolved model snapshots are unrecorded.

Short name	Requested model identifier
Opus	`claude-opus-5`
Sonnet	`claude-sonnet-5`
Haiku	`claude-haiku-4-5-20251001`
Astra	`gpt-6-astra`

Table 29: Requested model identifiers recorded in the generation summaries.

Table 30 reports the generation settings and summary statistics for the retained programs: 41 tasks for each of the four models. The mixture-of-experts, recommender and sparse-convolution component tasks were added and prompted after results for the original tasks were available. Their programs were generated without execution feedback and enter the same evaluation as the other programs. Opus, Sonnet and Astra used the September 21 generation round for these tasks. Haiku's three component programs were generated on September 25 (UTC), with the same generator, system prompt, tool restrictions and task-prompt fingerprints used for Opus and Sonnet.

Generation time limits did not exceed 3600 seconds per call. All models receive exported task contracts and a formatting-only system prompt. The contracts include the reference implementation.

	Opus	Sonnet	Haiku	Astra
tasks attempted	41	41	41	41
programs retained per task	1	1	1	1
generation time, total (s)	9433	6034	3219	11849
generation time, longest (s)	857	413	132	558
recorded generation cost (USD)	24.35	7.84	2.86	unavailable

Table 30: Generation settings for the retained programs. Time and cost summaries cover only the calls that produced these programs.

All calls use empty temporary working directories. Claude uses an empty configuration directory with tool interfaces disabled. Astra ignores user configuration and disables execution, delegation, app and browser features through Codex. Its generation records contain no recorded tool calls. Astra requests `xhigh` reasoning effort and records 1,484,076 input and 383,517 output tokens, including 299,763 reasoning output tokens. Its dollar cost was not recorded.

Sampling and software configuration. Apart from Astra's requested reasoning effort, sampling settings and output-token limits are unrecorded. The generator leaves them at client defaults. The generation prompts specify the GPU and its compute capability, but not the installed framework or compiler versions. The evaluation environment below was therefore not fully specified to the models.

Evaluation environment. We use NVIDIA L40S GPUs (compute capability 8.9), PyTorch 2.11.0+cu128, cuSPARSE 12.5.8, Triton 3.6.0, Python 3.12.3, NumPy 2.5.3 and CUDA 12.8.

F Supporting analyses

F.1 Input provenance and analysis design

Input provenance. All four models use the same case specifications: 533 performance and 637 correctness-only cases. The generation prompts disclose 328 public correctness-case descriptions, including shapes, generators, index dtypes and seeds. All 328 match evaluated correctness cases and constitute public conformance examples. Performance cases, actual operand arrays and some generator parameters are omitted from the prompts. Prompt hashes match the generation records for all four models. Among the three programs that fail correctness-only inputs after passing all performance checks, Astra's convolution fails a case described in its prompt, while the Opus GAT and Astra GCN failure cases are absent from their prompt examples. All three cases are public benchmark inputs. Of these descriptions, 304 come from the original 38 tasks and eight from each of the three component tasks. For the original tasks, the case specifications precede the committed copies of the evaluated programs, including the three correctness-only cases that reject programs passing every performance input. Incomplete generation timestamps limit this chronology to the committed artifacts.

Exploratory analyses. The comparisons on inputs shared by both programs were chosen after inspecting results and are exploratory (Nosek et al., 2018; Bertinetto et al., 2021). The internal analysis plan was not externally registered. Comparisons of means over different input sets can confound performance with the inputs each program supports. The reported runtime comparisons therefore use the same inputs for both programs (Appendix C.6). The plan and its amendments are available in `paper/PREREGISTRATION.md`.

F.2 Controlled sparse-structure comparison

Table 31 compares the Opus programs and their library baselines for embedding-bag sum, mean and max under a change in row-length skew. Each task uses two performance inputs with shape $16{,}384 \times 1{,}048{,}576$, exactly 524,288 nonzeros, 64 features, float32 values and int32 indices. Structure and value seeds are fixed at 133 and 233, with only the generator's skew parameter changing. The dense

embedding table is identical. Table 31 reports the structural statistics of these inputs alongside their measured runtimes.

Input property		Skew 1.5	Skew 3
Nonzeros		524,288	524,288
Mean row length		32	32
Longest row		4,825	72,082
Empty rows (%)		2.15	53.81
Row-length standard deviation / mean		3.015	23.285
Mean column span of nonempty rows		757,645	491,986
Task	**Implementation**	**Time (ms)**	**Time (ms)**
Embedding-bag sum	Dedicated PyTorch entry point	1.522	17.834
	PyTorch CSR matrix product	**0.257**	**0.255**
	Opus program	0.441	3.641
Embedding-bag mean	Dedicated PyTorch entry point	1.504	17.526
	PyTorch CSR matrix product	**0.271**	**0.270**
	Opus program	0.729	8.257
Embedding-bag max	Dedicated PyTorch entry point	1.177	13.080
	PyTorch scatter-reduce	1.584	4.929
	Compiled scatter-reduce	1.571	**4.920**
	Opus program	**0.638**	6.655

Table 31: **Input properties and embedding-bag runtimes.** Input properties and device times at fixed dimensions, nonzero count, feature width and dtypes. Bold marks the fastest implementation for each task and input. Column span is the largest minus the smallest column index plus one. Each timing cell uses one recorded measurement.

Table 32 gives the max comparison for all three models on the same original seed pair, including compiled scatter-reduce. Opus and Sonnet each go from faster than all three tested baselines to slower than scatter-reduce, while Astra remains faster at both settings. The comparison holds size and density fixed while changing sparse structure. It uses one seed pair and one selected measurement per case. Row imbalance, empty-row fraction and column locations change together in this comparison.

On the sum task at skew 3, the dedicated PyTorch baseline takes 17.834032 ms, its CSR matrix-product alternative takes 0.254976 ms, and the candidate takes 3.641024 ms. The same candidate therefore has a speedup of $4.898\times$ against the first baseline and $0.070\times$ against the second. The comparison concerns two implementations within one library. The matrix-product formulations remain approximately constant across the two measured skew settings, while the dedicated entry points slow by about 11–12-fold. The max task's scatter-reduce alternative also slows, by about $3.1\times$ on this seed pair.

Replication with fresh inputs. We repeat the max comparison on eight fresh seed pairs at the same dimensions, nonzero count, feature width and dtypes. Within each pair, values and the dense embedding table match across skew 1.5 and skew 3. Each input runs in two separate timing passes. These independent measurements compare the dedicated embedding-bag operator and the eager scatter-reduce composition, without compiled variants. A faster or slower classification requires the speedup against the faster of those two baselines to lie outside $[1/1.05, 1.05]$ in the same direction in both passes, with the recorded timing-quality checks satisfied.

Opus and Sonnet outperform both of these baselines at skew 1.5 in all eight pairs. Their original crossover reproduces in four pairs. In three they remain faster at skew 3, and in one the skew-3 result falls within the margin. Astra remains faster at both settings in all eight pairs. The baseline ordering also depends on the pair: embedding-bag is faster at skew 1.5 in all eight, while at skew 3 scatter-reduce is faster in six, embedding-bag is faster in one, and the two differ by less than 2% in one. Thus the crossover recurs on new inputs, but not on every input pair at these skew settings. The replication was designed after observing that example and is separate from the structural-coverage comparison in Appendix C.14. Haiku's program lacks a custom kernel and is rejected by the source screen on all sixteen inputs in both passes, so it has no runtime comparison.

Model	Skew	Median runtime (ms) Generated program	PyTorch embedding-bag	PyTorch scatter-reduce	Scatter-reduce `torch.compile`	Speedup vs. fastest baseline
Opus	1.5	**0.638**	1.177	1.584	1.571	1.84×
	3	6.655	13.080	4.929	**4.920**	0.74×
Sonnet	1.5	**0.626**	1.172	1.532	1.571	1.87×
	3	6.718	13.091	**4.909**	4.936	0.73×
Astra	1.5	**0.324**	1.176	1.534	1.570	3.63×
	3	**1.898**	13.089	4.896	4.924	2.58×

Table 32: **CSR embedding-bag max at two skew settings.** Both inputs have 16,384 rows, 1,048,576 columns, 524,288 nonzeros and 64 features, with the same dtypes and dense embedding table. Bold marks the fastest time in each row. Speedup compares the generated program with the fastest of the three baselines measured in the same run, one of them the scatter-reduce composition compiled with `torch.compile`. One seed pair, one evaluation per row. Appendix F.2 details the controls.

Program	Skew 1.5 Faster	Skew 3 Slower	Skew 3 Faster	Skew 3 Undecided
Opus	8	4	3	1
Sonnet	8	4	3	1
Astra	8	0	8	0

Table 33: **Replication of the embedding-bag max example on eight fresh seed pairs.** Counts compare each program with the faster of the dedicated embedding-bag operator and the eager scatter-reduce composition on the same input. Compiled variants are not measured in this independent replication. Faster/slower requires the same direction outside the interval $[1/1.05, 1.05]$ in both independent timing passes. The original crossover, from faster at skew 1.5 to slower at skew 3, occurs in 4 of 8 pairs for Opus and 4 of 8 for Sonnet. Dimensions, nonzero count, width, values and the dense embedding table are matched within each pair.

F.3 Baseline choice and timing variability

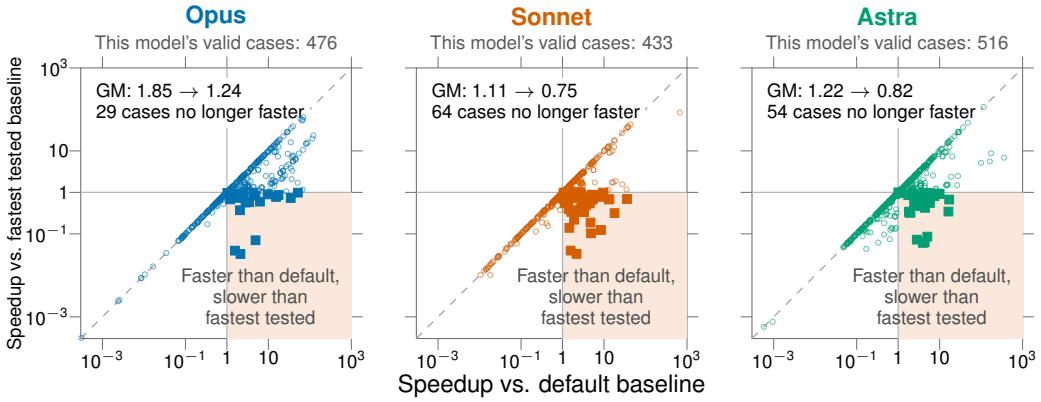


Figure 9: **A program can be faster than one baseline and slower than another.** This diagnostic uses all 41 evaluated tasks and 533 performance cases, retaining 476, 433 and 516 valid speedups for Opus, Sonnet and Astra from programs passing source review. Both coordinates of a point use the same candidate runtime. Squares mark 29, 64 and 54 cases faster than the default baseline but slower than the fastest tested one. Circles stay on the same side of parity. The diagonal marks equal baseline times. Annotations give geometric-mean speedups against the default and fastest tested baselines, respectively. Each panel's geometric means use that model's valid cases. Table 1 uses cases valid for all three models.

Figure 9 compares speedups against the default baseline on the horizontal axis and the fastest tested baseline on the vertical axis. Using default rather than fastest baselines multiplies Sonnet's speedups on its 433 valid cases by a geometric-mean factor of 1.48 (95% bootstrap interval over cases: 1.38–1.60).

We resample these paired default/fastest ratios with replacement 10,000 times (seed 20260920), taking the 2.5th and 97.5th percentiles of their geometric means. This interval measures variation across the recorded cases, with each case's measured runtime held fixed.

Sensitivity to close baseline timings. Using Opus programs that passed the automated source screen, before applying source-review exclusions, we examine 1,268 baseline measurements across 428 cases with multiple accepted baselines. Absolute relative differences between repeated timings of the same implementation have median 0.000490 and maximum 0.092120. On 31 cases, the relative gap between the fastest and second-fastest baselines is no greater than their larger replicate deviation. For these 31 cases, we substitute the default baseline's runtime for the fastest baseline's runtime in this sensitivity calculation. Their ratios of default to selected baseline runtime are set to 1, changing the geometric mean across all 428 cases from 1.5673 to 1.5634. This analysis uses repeated timings within each run.