DataKernelBench: Can LLMs Optimize Database Queries on GPUs?

Gokul Karthik Kumar[1,2], Yotam Perlitz[1], Corey Lammie[1],
Andrea Giovannini[1], and Katja Hose[2]

[1]IBM Research, Zurich, Switzerland
[2]TU Wien, Vienna, Austria
gok@zurich.ibm.com, y.perlitz@ibm.com, corey.lammie@ibm.com,
agv@zurich.ibm.com, katja.hose@tuwien.ac.at

Abstract

GPUs increasingly accelerate database systems, but query-specific peak performance still often relies on hand-written kernels. Existing LLM kernel benchmarks focus on machine learning operators, leaving irregular, heterogeneous, data-movement-heavy database-style operators untested. We introduce **DataKernelBench**[1], which translates SQL into validated PyTorch **TorchPlan** programs and evaluates LLMs that optimize either the core tensor-bounded snippet or the full query in CUDA or Triton through execution-guided repair. Across ten proprietary and open-weight models on **TPC-H SF10** with an H100 GPU, the strongest full-query CUDA configuration achieves **2.11×** **speedup over** `torch.compile` at full pass rate. We find that higher-performing implementations commonly use kernel fusion and execution-strategy changes, stronger models benefit most from full-query specialization, and workload context matters more than hardware context. To handle data larger than GPU memory, we extend TorchPlan with Dask-cuDF for on-demand partition loading on **TPC-H SF100** with four H100 GPUs, achieving **2.54×** speedup.

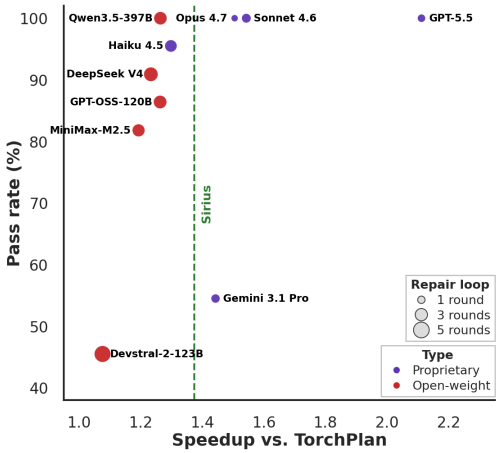

Figure 1: DataKernelBench tests LLMs on GPU kernel generation for database query optimization, where multiple models achieve 100% pass rate and substantial speedups over the TorchPlan baseline on TPC-H SF10 with NVIDIA H100. See Table 1 and Section 4.2.

1 Introduction

Massive AI infrastructure investments (International Data Corporation (IDC), 2025) have made GPUs increasingly common in enterprise environments (Gartner, Inc., 2025). At the same time, many analytical workloads have execution patterns that can benefit from GPU hardware: large scans, predicate evaluation, joins, and aggregations over columnar data expose substantial data parallelism and place heavy demand on memory bandwidth. This combination makes GPU acceleration an increasingly attractive direction for analytical data processing (Yogatama et al., 2026), especially for frequently executed queries whose cost can justify query-specific optimization.

One practical approach is Tensor Query Processing (TQP) (He et al., 2022; Asada et al., 2022), which maps relational operators such as joins, filters, and aggregations to tensor computations so that database workloads can run on mature machine learning (ML) frameworks such as PyTorch. Recent work shows that this approach supports terabyte-scale analytics on modern multi-GPU systems (Wu et al., 2025). These trends raise a natural question: if analytical queries can be expressed as tensor programs, *can latest code LLMs synthesize specialized GPU kernels that outperform generic compilation?*

This question matters because generic ML compilers often fall short of peak performance on analytical workloads. Although `torch.compile` is effective for regular, compute-bound neural operators, analytical queries often involve irregular memory access, complex predicates, heteroge-

[1] https://kerneldf.github.io/datakernelbench

neous operator combinations, and query-specific fusion opportunities. Peak performance therefore still often requires hand-written bespoke GPU kernels, yet writing and maintaining them is expensive even when specialization pays off for frequently executed reporting queries (Marcus, 2023; Wehrstein et al., 2026).

Recent progress in LLMs (OpenAI, 2023; Bai et al., 2022; Qwen Team, 2026a; Team, 2024; Jain et al., 2024; Liu et al., 2024) suggests a promising path toward reducing this engineering burden. Prior work shows that LLMs can generate specialized CUDA and Triton kernels for hardware acceleration (Woo et al., 2025; Dai et al., 2026; Liao et al., 2025; Lange et al., 2025; Baronio et al., 2026), and recent benchmarks such as KernelBench (Ouyang et al., 2025), TritonBench (Li et al., 2025), and MultiKernelBench (Ouyang et al., 2025) evaluate this capability on ML operators. However, these benchmarks focus on regular tensor computations with predictable structure. Analytical queries can instead involve substantial data movement, combine multiple relational operators, and require strict correctness under data-dependent control flow. Strong performance on ML-centric kernel synthesis benchmarks therefore does not establish that LLMs can handle database-style workloads.

To address this gap, we introduce **DataKernelBench** for evaluating LLM-generated GPU kernels for analytical query processing. For each query, we first translate SQL into a validated PyTorch tensor program, which we call **TorchPlan**. TorchPlan serves as a benchmarkable intermediate representation: it preserves SQL semantics while exposing a stable optimization target for kernel synthesis. It separates table handling in `run_query` from the tensor-intensive hot path in `_query_core`, enabling controlled specialization at two levels: `core`, where only the hot path may be replaced, and `full`, where the full internal query implementation may be rewritten while preserving the external API. Each baseline TorchPlan is validated by comparing its outputs to DuckDB (Raasveldt and Mühleisen, 2019) before kernel generation. We then ask LLMs to inject optimized Triton or CUDA kernels under a multi-round execution-guided repair loop.

Our primary baseline is compiled TorchPlan execution, since TorchPlan provides a validated tensor reference and `torch.compile` is the strongest generic baseline within this execution model. The benchmark therefore measures the incremental gain of bespoke LLM-generated kernels over a strong compiled tensor baseline. We also compare against external systems such as Sirius (Yogatama et al., 2026) and DuckDB, but these answer a different systems question: *how fast does an independent execution engine run the workload?* Sirius is best viewed as a generic GPU-specialized database system that provides drop-in GPU acceleration for existing CPU database engines, whereas our method targets bespoke kernels for specific, frequently executed queries.

DataKernelBench targets recurring queries such as scheduled reports and dashboard refreshes, where the same query template runs on refreshed data, with different user-supplied parameters, or both. Before specialization, a deployment can compare predicted generation cost with the expected reduction in execution time or compute cost over future runs. A generated kernel is adopted only after validation and when the expected savings justify specialization; otherwise, execution falls back to compiled TorchPlan or a general-purpose GPU query engine. Table 1 reports one such runtime threshold through N_{save1h}. As one possible integration path, a generated CUDA kernel could be wrapped as a GPU operator in an extensible engine such as Velox's experimental cuDF backend, whose `DriverAdapter` supports operator replacement, fusion, and addition (Velox, 2022).

With DataKernelBench, we evaluate ten proprietary and open-weight LLMs across CUDA and Triton backends and both optimization levels (Section 4). Gains are highly query-dependent, and the best LLM-optimized kernels complement rather than uniformly replace a generic GPU database system. Plan inspection attributes the largest gains to kernel fusion and execution-strategy changes that avoid intermediate materialization. We also provide preliminary evidence beyond a single GPU's memory by executing TPC-H SF100 in on-demand Dask-cuDF partitions distributed across four H100 GPUs (Section 4.5).

This work makes three contributions. First, we introduce, to the best of our knowledge, the first systematic benchmark for LLM-generated GPU kernels for database queries, as opposed to ML operators. Second, we provide a validated SQL-to-TorchPlan pipeline and modular Python TorchPlan programs that expose an LLM-friendly optimization target and serve as pluggable reference

implementations for kernel synthesis. Third, our evaluation of ten LLMs characterizes how model strength, optimization scope, prompt context, programming interface, and query structure affect correctness and performance, and uses plan-level comparisons to explain where the largest gains arise.

2 Related Work

2.1 Databases, GPUs, and Analytical Processing

GPU-accelerated analytical processing has been pursued through modular libraries and full database systems. Libraries such as `libcudf` and RAPIDS cuDF (RAPIDS, 2023) accelerate dataframe-style workloads (Pandas, 2020), while composable engines such as Velox (Velox, 2022) explore how specialized operator libraries can be integrated into flexible pipelines. Other work develops GPU-specialized database systems such as Sirius (Yogatama et al., 2026), Kinetica (Kinetica, 2026), and SQream (SQream Technologies, 2026). At a lower level, Kernel Weaver (Wu et al., 2012) shows that automatically fused GPU kernels can substantially reduce redundant data movement in relational workloads. These results reinforce the value of specialization for analytical GPU processing, but they operate at the level of a full execution system, compiler framework, or fixed relational primitives. In contrast, our focus is to evaluate whether LLMs can synthesize *bespoke kernels* for analytical queries within an existing tensor-based query pipeline.

Our work is closest in spirit to Tensor Query Processing (TQP) (He et al., 2022; Asada et al., 2022), which maps relational operators to tensor computations on ML runtimes. Subsequent work narrows the mismatch between SQL operators and tensor operations (Zhang et al., 2025) and supports execution directly on compressed data (Huang et al., 2025). Recent work also shows that tensor-based query processing can scale to terabyte-scale multi-GPU analytics (Wu et al., 2025) and distributed, storage-resident OLAP (Luo et al., 2025). However, compiled tensor execution does not eliminate the need for specialization: analytical queries often involve irregular access patterns, heterogeneous operators, and query-specific fusion opportunities that expose a generalization ceiling in generic compilation. DataKernelBench is designed to evaluate whether LLM-generated kernels can close this gap.

2.2 AI and GPUs for Kernel Synthesis

Recent progress in code LLMs has led to advances in automated kernel generation for hardware acceleration (OpenAI, 2023; Bai et al., 2022; Qwen Team, 2026a; Zeng et al., 2025). Systems such as TritonRL (Woo et al., 2025), CUDA Agent (Dai et al., 2026), and KernelEvolve (Liao et al., 2025) iteratively synthesize and optimize CUDA or Triton kernels, showing that LLM-based kernel generation can be a viable path toward hardware specialization. Several benchmarks now evaluate this capability systematically. KernelBench (Ouyang et al., 2025), TritonBench (Li et al., 2025), and MultiKernelBench (Wen et al., 2025) measure pass rate and performance for LLM-generated kernels across hardware targets and programming abstractions.

These benchmarks are an important methodological foundation for our work, but they are centered on machine learning operators, where computations are typically dense, regular, and dominated by homogeneous tensor primitives. Analytical query processing instead combines heterogeneous relational operators with irregular data access and strict output semantics. DataKernelBench builds on the evaluation style of these prior benchmarks while shifting the target domain from ML kernels to database-style analytical workloads.

2.3 AI and Data Systems

AI has long been applied to data systems, first by replacing hand-designed heuristics with learned components such as learned query optimizers (Marcus et al., 2019) and learned indexes (Kraska et al., 2018). With the rise of LLMs, AI in data systems has expanded from user-facing interfaces such as text-to-SQL (Li et al., 2024; Gao et al., 2024) toward automated system synthesis. The closest line of work to ours is Bespoke OLAP (Wehrstein et al., 2026), which synthesizes workload-specific database engines to outperform general-purpose systems such as DuckDB (Raasveldt and Mühleisen, 2019). Our work differs in both scope and interface: rather than generating a standalone database engine, we study whether LLMs can synthesize *pluggable GPU kernels* that accelerate specific bottlenecks within validated TorchPlan pipelines.

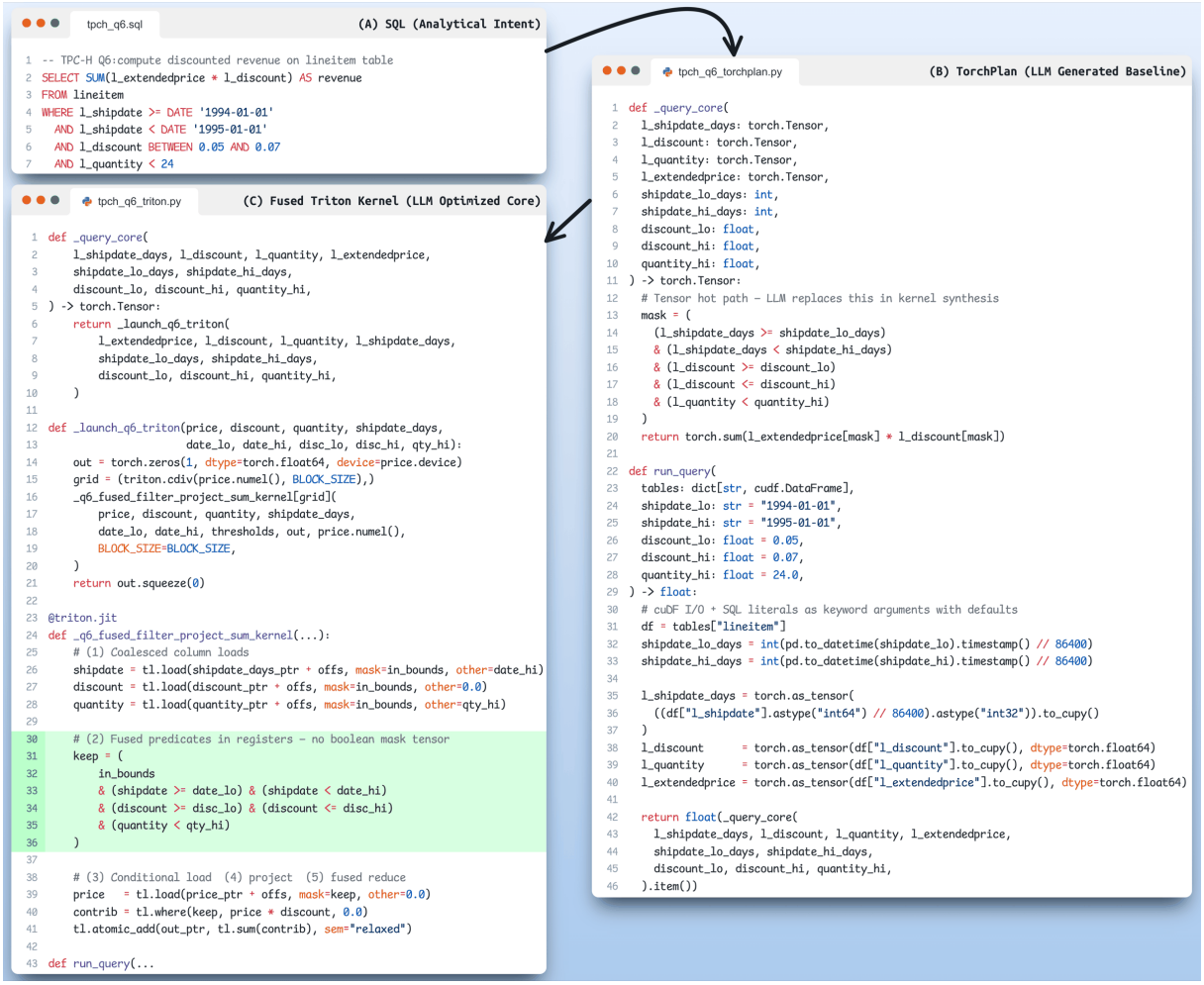

Figure 2: From SQL to TorchPlan to an LLM-synthesized fused GPU kernel for TPC-H Q6. **(A)** SQL with parameterized literals. **(B)** TorchPlan, where `run_query` handles cuDF table processing and `_query_core` defines the tensor hot path optimized by `torch.compile` in the baseline. **(C)** An LLM-generated fused Triton implementation at the `core` level. The analogous CUDA example appears in Figure 5 of Appendix A; the Dask-cuDF variant of (B), as described in Section 4.5, appears in Figure 6 of Appendix B.

3 The DataKernelBench Framework

DataKernelBench instantiates LLM-based kernel synthesis for analytical query processing as a two-stage benchmark pipeline: (1) construct and validate a baseline **TorchPlan** for each query, and (2) ask LLMs to inject optimized Triton or CUDA kernels that preserve semantics while improving runtime. Figure 2 illustrates this SQL→TorchPlan→kernel mapping for TPC-H Q6. While Q6 is a simple single-table query that we chose to illustrate our approach, our benchmark covers all 22 TPC-H queries, including joins, nested subqueries, and more complex hot paths. This design cleanly separates benchmark construction from model evaluation: TorchPlans define validated task instances, while kernel synthesis measures how effectively LLMs optimize those instances under fixed pass rate and runtime constraints. The supplementary material includes the prompts, validated TorchPlans, and selected optimized CUDA and Triton implementations. We will release these artifacts and the evaluation harness at `https://github.com/kerneldf/datakernelbench`.

3.1 Benchmark Workload and Setting

Our primary evaluation uses **TPC-H** (Transaction Processing Council, 2018) at **scale factor 10** which contains 22 analytical queries over 8 relational tables. We generate benchmark artifacts using DuckDB's `tpch` extension, including data, SQL queries, and reference outputs. At runtime, tables are loaded into `cuDF` (RAPIDS, 2023) dataframes so that `run_query` executes on GPU-resident data.

3.2 TorchPlan: A Benchmarkable Intermediate Representation

Each query is first translated by an LLM into a **TorchPlan**, an executable PyTorch tensor program that serves as the benchmark's intermediate representation (IR) within the TQP paradigm. TorchPlan preserves SQL semantics while exposing a stable optimization target for kernel synthesis. In this sense, TorchPlan acts as a *verified contract* between declarative query logic and imperative GPU kernel generation: it is directly executable, amenable to differential testing against a reference engine, and compatible with multiple optimization backends. We do not propose a new TQP system; rather, we use validated TQP-style tensor programs as a strong reference representation and execution substrate, then measure how much additional performance bespoke LLM-generated kernels can obtain beyond compiled tensor execution. Each TorchPlan contains two functions:

- `run_query(tables: dict[str, cudf.DataFrame], ...)`, which handles table access, joins, projections, parameter parsing, and output formatting; and
- `_query_core(...)`, which implements the tensor-intensive hot path over aligned 1D GPU tensors and Python scalars.

This decomposition mirrors the TQP design: relational preparation remains in the dataframe layer, while the tensor-intensive hot path is isolated in `_query_core` for specialization. TorchPlan also reflects realistic analytical execution, where the same query template may be run repeatedly with different literals. Accordingly, `run_query` exposes SQL literals as keyword arguments with defaults, while `_query_core` receives only aligned GPU tensors and numeric scalars. Figure 7 in Appendix C provides a representative multi-join example that makes this boundary concrete.

We generate baseline TorchPlans with a single LLM (Claude Opus 4.7 in our main setup) using a fixed prompt template provided in the supplementary material. We then validate each generated plan by executing the original SQL in DuckDB and comparing the result to `run_query` on the same input tables. Only validated TorchPlans enter the benchmark, and each is held fixed across all model, framework, and optimization-level evaluations. These TorchPlans serve as both the correctness reference for synthesized kernels and the baseline execution path from which speedup is measured. To assess how the TorchPlan generator affects final optimization, Appendix D repeats the evaluation with GPT-5.5-generated TorchPlans. The key findings persist, although absolute runtimes change.

3.3 Kernel Synthesis Task

Given a validated TorchPlan, the benchmark asks an evaluated LLM to produce a faster implementation that preserves the external `run_query` semantics. We study two optimization levels:

- `core`: the model may modify only `_query_core` and private helper functions; `run_query` must remain unchanged. This keeps the optimization target close to ML kernel benchmarks, whose inputs and outputs are tensors and scalars rather than complex datatypes (e.g., String and DataFrames);
- `full`: the model may rewrite `run_query`, `_query_core`, and helpers, while preserving the external `run_query` API.

We support two GPU programming interfaces, **Triton** and **CUDA**. In both cases, the model must return one complete Python module exposing `run_query`. This turns kernel synthesis into a constrained program-rewriting task: the model is free to optimize the internal computation, but not to alter the benchmark-facing semantics.

3.4 Prompt Design

Kernel-synthesis prompts are composed from six markdown blocks concatenated in fixed order: BASE, LEVEL, FRAMEWORK, GPU, ENV, and DATA. The BASE block defines task constraints, interface rules, and inserts the reference TorchPlan and original SQL. LEVEL specifies the allowed optimization scope. FRAMEWORK provides Triton- or CUDA-specific guidance and a worked fused-kernel example. GPU and ENV describe the target hardware and runtime environment. DATA provides workload metadata such as table names, dtypes, row counts, and approximate memory footprint at scale factor 10. This modular design supports controlled ablations, since workload, hardware, and framework context can be removed independently.

3.5 Correctness and Speedup Criteria

For each generated module, we compare the output of `run_query` against the baseline TorchPlan

on the fixed scale factor 10 tables. We report mismatches hierarchically: shape, column names and order, row count, and then cell-level differences with sampled offending values. Non-floating-point columns (e.g., integers, strings, and dates) must match exactly. Following prior kernel-benchmark practice (Ouyang et al., 2025), floating-point columns are compared using `numpy.isclose` with `atol=rtol=`10^{-4} for `float32` and 10^{-6} for `float64`. Failed checks are returned to the model as repair feedback.

We measure end-to-end `run_query` runtime. Before timed measurement, we discard N_{warmup} warmup runs to amortize compilation and GPU warm-start effects. Let T_{base} denote the median runtime of the validated TorchPlan when `_query_core` is wrapped with `torch.compile`, and let T_{cand} denote the median over N_{timed} timed runs of the generated module after warmup. We define speedup as $T_{\text{base}}/T_{\text{cand}}$. A generated module is accepted only if it is functionally correct and achieves speedup $\geq s_{\text{min}}$. We require both correctness and a minimum speedup so that accepted modules strictly improve on compiled TorchPlan; otherwise the baseline remains preferable.

3.6 Execution-Guided Generation

We evaluate every combination of TPC-H query, model, framework (Triton or CUDA), and optimization level (`core` or `full`) as an independent run. For each combination, we execute a multi-round repair loop with at most R_{max} rounds. In round 1, the model receives the composed prompt from Section 3.4 and returns one candidate Python module exposing `run_query`. Each later round appends feedback from the previous candidate, including validation failures, output mismatches, tracebacks, and measured speedup, and asks the model to produce a revised candidate. We stop at the first candidate that passes the correctness checks in Section 3.5 and satisfies speedup $\geq s_{\text{min}}$, or when R_{max} is reached. To bound hung executions, correctness checks (Section 3.5) and warmup runs are terminated after $\max(T_{\text{min}}, M_{\text{warmup}} \cdot T_{\text{base}})$, while each timed run is terminated after $M_{\text{timed}} \cdot T_{\text{base}}$. Defaults are: R_{max}=10, s_{min}=1.05, N_{warmup}=2, N_{timed}=5, M_{warmup}=20, M_{timed}=3, T_{min}=1 minute, and T_{wall}=15 minutes.

4 Experiments

4.1 Experimental Setup

We evaluate ten LLMs spanning proprietary APIs and open-weight models: GPT-5.5 (OpenAI, 2026), Claude Opus 4.7 (Anthropic, 2026a), Claude Sonnet 4.6 (Anthropic, 2026b), and Claude Haiku 4.5 (Anthropic, 2025), Gemini 3.1 Pro (DeepMind, 2026), Qwen3.5-397B-A17B (Qwen Team, 2026b), GPT-OSS-120B (OpenAI, 2025), DeepSeek-V4-Flash (DeepSeek-AI, 2026), MiniMax-M2.5 (AI, 2026), and Devstral-2-123B (Rastogi et al., 2025). All open-weight models were served on a separate machine. For each model, we evaluate CUDA and Triton generation, on an H100 80GB GPU, at both `core` and `full` levels, yielding 40 model–framework–level combinations. We use the default generation settings for each API or model checkpoint (e.g., temperature and top-p).

Within TorchPlan, `torch.compile` (TC) is the primary generic baseline, and all LLM speedups are measured against it; we also report TorchPlan eager execution to contextualize the strength of TC. For LLM-generated kernels, speedup and runtime use fallback to TC: if a generated kernel is functionally incorrect or fails to meet the minimum speedup threshold, that query is credited at the TC runtime, since such a kernel would not be adopted in practice. We also report Sirius and DuckDB as external DB baselines. Because these are independent execution engines rather than in-pipeline replacements, we always report their observed runtimes directly, including cases below parity with TC. We report the hardware and software environment details for benchmark execution, in Appendix E.

4.2 Comparative Evaluation of LLMs

Table 1 presents the main leaderboard on TPC-H SF10 on H100. The strongest result is GPT-5.5 with CUDA at the `full` level, which achieves **100%** pass rate and **2.11**× overall speedup over TorchPlan-`compile`. This also outperforms Sirius at **1.37**×. Among open-weight models, Qwen3.5-397B-A17B with Triton-`full` is strongest at **1.26**× with **100%** pass rate, while GPT-OSS-120B matches the same speedup with lower pass rate.

The leaderboard shows a clear relationship between model quality and synthesis efficiency. Top-ranked models generally require fewer repair rounds and fewer tokens to reach their best configuration, whereas weaker models spend more on un-

Model	Best	Pass	↑ Speedup vs. TorchPlan				↓ Runtime		↓ Mean Generation Cost			
	Config	Rate%	Overall	≥1.05×	≥1.20×	≥2×	Overall (s)	N_{save1h}	R	In (k)	Out (k)	USD
Proprietary models												
GPT-5.5	CUDA-full	**100.0**	**2.11**	**100.0**	**95.5**	**45.5**	**0.71**	**4535**	1.4	**12.6**	9.8	0.18
Claude Sonnet 4.6	Triton-full	**100.0**	1.54	95.5	77.3	13.6	0.98	6786	1.9	26.0	6.2	0.13
Claude Opus 4.7	CUDA-full	**100.0**	1.51	**100.0**	81.8	13.6	1.00	7116	**1.0**	14.0	**4.7**	0.14
Gemini 3.1 Pro	CUDA-full	54.5	1.44	50.0	45.5	18.2	1.04	7775	1.7	13.0	14.9	0.21
Claude Haiku 4.5	Triton-full	95.5	1.30	90.9	68.2	9.1	1.16	10394	3.2	59.9	9.1	**0.08**
Open-weight models												
Qwen3.5-397B-A17B	Triton-full	**100.0**	1.26	86.4	72.7	4.5	1.19	11432	3.7	58.8	16.9	—
GPT-OSS-120B	CUDA-full	86.4	1.26	81.8	72.7	4.5	1.19	11484	3.8	69.7	14.1	—
DeepSeek-V4-Flash	Triton-core	90.9	1.23	86.4	63.6	0.0	1.22	12633	4.4	80.9	9.7	—
MiniMax-M2.5	Triton-full	81.8	1.19	81.8	68.2	4.5	1.26	14786	3.5	50.4	15.9	—
Devstral-2-123B	Triton-full	45.5	1.08	36.4	27.3	0.0	1.40	33782	5.9	124.2	12.0	—
Primary baselines												
TorchPlan (compile)	—	100.0	1.00	0.0	0.0	0.0	1.51	—	—	—	—	—
TorchPlan (eager)	—	100.0	0.88	13.6	0.0	0.0	1.71	—	—	—	—	—
DB baselines												
Sirius (GPU)	—	100.0	1.37	68.2	63.6	54.5	1.10	8765	—	—	—	—
DuckDB (CPU)	—	100.0	0.54	9.1	9.1	4.5	2.81	—	—	—	—	—

Table 1: DataKernelBench leaderboard results on TPC-H SF10 using an NVIDIA H100, reporting the best kernel configuration per model. **Pass Rate%** denotes the percentage of queries matching the reference output. **Speedup** measures performance relative to TorchPlan with `torch.compile` (TC), where the threshold columns indicate the fraction of the 22 queries achieving at least $1.05\times$, $1.20\times$, and $2\times$ speedups. $\mathbf{N}_{\text{save1h}}$ represents the number of repetitions required to save 1 hour of execution time relative to TC. **R** is the number of rounds used to get final generation. **In (k)** & **Out (k)** denotes token usage in thousands; **USD** is the API usage cost. Arrows indicate higher (↑) or lower (↓) values are better. The best results are **bolded** and second-best are underlined.

successful repairs. For example, GPT-5.5 reaches the top result with only **1.4** mean rounds, while lower-ranked open models require **3.5–5.9** rounds. At the same time, token cost alone is not a sufficient proxy for quality: Gemini 3.1 Pro reaches **1.44**× speedup with moderate token cost, but passes only **54.5%** of queries. Appendix F analyzes all 880 generation trajectories (22 queries × 10 models × 2 frameworks × 2 optimization levels), including repair progression, failure modes, and query difficulty.

Backend preference is model-dependent. Triton is the best configuration for six of the ten models, while CUDA is best for four. Interestingly, both OpenAI models achieve their best results with CUDA, despite OpenAI's central role in Triton's development. A plausible interpretation is that Triton is easier to synthesize reliably, which benefits weaker models, while stronger models are able to exploit the higher performance ceiling of CUDA.

4.3 Additional Correctness Validation

The primary pass rate uses output matching on SF10 data, as defined in Section 3.5. We further subjected all 22 GPT-5.5 CUDA-`full` implementations, the top-performing configuration, to two complementary robustness checks beyond this protocol: source-code audits and held-out differential testing.

For the source audits, GPT-5.5 produced two verdicts for each implementation. `semantic_match` checks predicates, join and group-by keys, aggregation formulas, date and null handling, output schema, and parameter use. `no_cheating` checks for hard-coded outputs, cached or reference answers, and bypassed input tables. For held-out testing, we used SF1 and a perturbed SF10 dataset that preserves keys and text identifiers while applying up to $\pm 30\%$ jitter to numeric columns, shifting dates by up to ± 30 days, and resampling low-cardinality values from their existing domains. **All 22 implementations passed both source-audit verdicts and both held-out validations**. These checks provide evidence beyond output matching on the original SF10 data, although they do not constitute a formal proof of semantic equivalence.

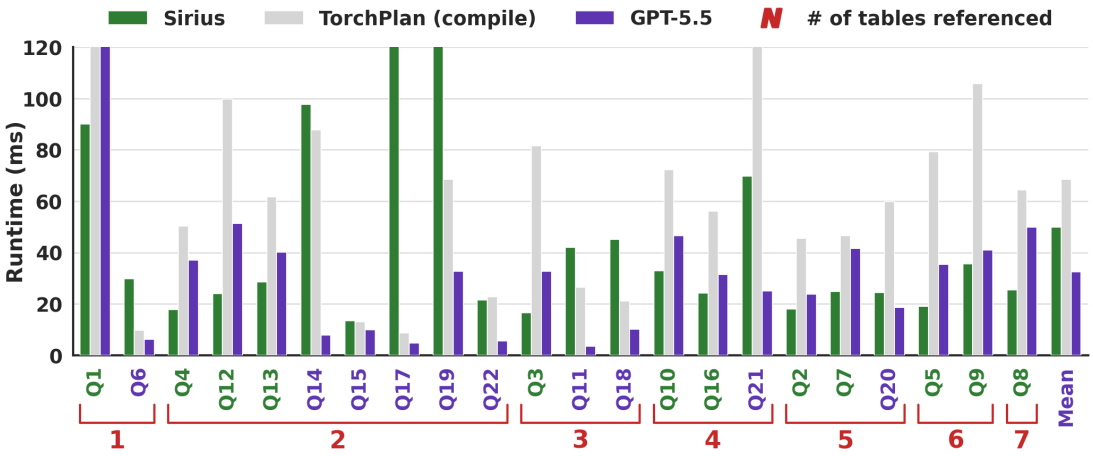

Figure 3: Per-query runtime comparison: GPT-5.5 CUDA at Full level vs. baselines. Y-axis capped at 120ms.

4.4 Understanding Performance Gains

Figure 3 shows per-query runtime for TorchPlan-`compile`, Sirius, and GPT-5.5 CUDA-`full`. The main pattern is strong non-uniformity: Sirius is fastest on many queries, while GPT-5.5-generated kernels dominate on a different subset. The advantage narrows as queries reference more tables. GPT-5.5 is faster than Sirius on **8/13** queries referencing at most three tables, but only **2/9** queries referencing more than three, while overall runtime still favors GPT-5.5. This pattern is consistent with Appendix G: all ten LLMs show negative correlation between choke-point coverage and speedup, whereas Sirius shows positive correlation.

To identify where the speedups arise, we compared GPT-5.5 CUDA-`full` implementations with their compiled TorchPlan baselines across all 22 queries. The dominant pattern is **kernel fusion**: filtering, projection, and aggregation are combined into one or a few passes, avoiding intermediate masks, gathers, and temporary tensors. **Hybrid execution** is also common: 21/22 implementations retain cuDF for string or DataFrame operations and use CUDA for the numeric hot path. In addition, 14/22 replace `torch.unique/scatter_add` group-by pipelines with **fused CUDA aggregation**.

Q14 provides the clearest example of a broader plan rewrite. GPT-5.5 builds a compact lookup of promotional `part` keys and probes it while scanning `lineitem`, avoiding a materialized join. This yields **11.2× speedup**, while CUDA-`full` alternatives retaining the cuDF merge remain near 1.07×. Similar advantages appear on Q17 and Q19. These results support a complementary interpretation: bespoke LLM-synthesized kernels do not replace generic GPU database systems on every query, but can deliver large wins on selected analytical pipelines that change the benchmark-wide ranking.

4.5 Beyond GPU Memory: Partitioned Multi-GPU Execution

We conducted a proof of concept for data that does not fit in GPU memory by replacing eager cuDF tables, which are loaded in full, with **partitioned Dask-cuDF** tables loaded on demand (Rocklin, 2015; RAPIDS, 2023). The extended `run_query` uses `map_partitions` to invoke `_process_partition` on each cuDF partition, which converts its columns to tensors and calls `_query_core`. Figure 6 in Appendix B shows this interface for TPC-H Q6. Dask distributes these partitions across the available GPU workers, enabling the same partitioned execution to use multiple GPUs.

We generated partitioned TorchPlans and GPT-5.5 CUDA-`full` implementations for **TPC-H SF100** and evaluated them on **four H100 80 GB GPUs** using 10 GB chunks. **All 22** optimized implementations passed the correctness checks. The complete workload finished in 36.44 s, a **2.54× speedup** over the corresponding Dask-cuDF TorchPlan baseline. Unlike the main SF10 evaluation, this timing includes on-demand data loading and materialization. These results provide preliminary evidence that the approach can extend to partitioned data larger than GPU memory; Appendix H reports how the same plans scale from 1 to 4 GPUs.

	CUDA	Triton
Top-5 models	+0.31	+0.29
Bottom-5 models	+0.02	+0.05

Table 2: Mean Speedup gains (Green denotes ≥ 0.10) from `full` vs. `core` optimization.

Config	Base	No GPU Prompt	No Data Prompt
CUDA	2.11	−0.27	−0.40
Triton	2.05	−0.06	−0.22

Table 3: Prompt ablation for GPT-5.5 (`full`) showing change in speedups. Red denotes a drop ≥ 0.10.

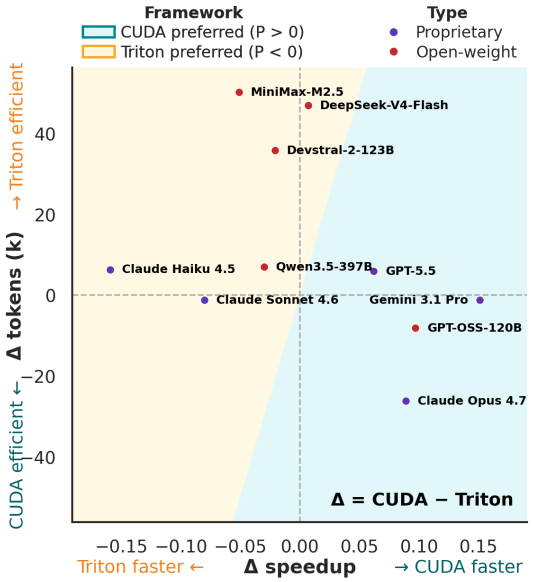

Figure 4: CUDA vs. Triton preference per model. Δspeedup and Δtokens (k) measure the runtime and token usage change between frameworks. The boundary separates CUDA-preferred ($P > 0$) from Triton-preferred ($P < 0$) regions at default $\lambda = 10$.

4.6 Effect of Optimization Scope: `core` vs. `full`

Table 2 compares `full` against `core` optimization by leaderboard tier. For the top-5 models, expanding synthesis from the tensor hot path to the full TorchPlan yields mean speedup gains of **+0.31** for CUDA and **+0.29** for Triton. For the bottom-5 models, the gains are only **+0.02** and **+0.05**.

This suggests that end-to-end query specialization is an emergent capability of stronger code LLMs rather than a uniform benefit across the leaderboard. Weaker models can sometimes optimize local tensor kernels, but they obtain little additional benefit from the broader optimization surface exposed by `full` generation. In contrast, stronger models are able to exploit opportunities that span table handling, launch logic, and fused execution, which helps explain why the leaderboard is led by `full`-level configurations.

4.7 Prompt Ablation

Table 3 reports prompt ablations for GPT-5.5 at the `full` level, comparing the complete prompt with variants that remove either hardware context from the GPU block or workload context from the DATA block. The results show that while hardware-aware prompting provides clear benefits, workload-aware prompting is significantly more critical. In both frameworks, omitting data characteristics degrades performance more severely than removing GPU details. This impact is most pronounced under CUDA, where excluding data properties triggers a substantial performance drop of **−0.40** compared to **−0.27** for GPU details. Triton exhibits a parallel trend, showing only a minor degradation (**−0.06**) when omitting GPU specifications, but a sharper drop (**−0.22**) without workload data. Thus, concrete data properties such as table sizes and types are more vital for guiding effective kernel optimization than raw hardware specifications.

4.8 CUDA vs. Triton Trade-offs

Figure 4 compares CUDA and Triton across models along two dimensions: runtime gain and generation cost. We define Δspeedup as the speedup difference between CUDA and Triton, and Δtokens as the corresponding difference in token usage measured in thousands. To combine them into a practical preference score, we use

$$P = \lambda\,(100 \cdot \Delta\text{speedup}) - \Delta\text{tokens},$$

where λ denotes the maximum additional thousand tokens a practitioner is willing to spend for a 1% speedup improvement. We use $\lambda = 10$ as the default; models with $P > 0$ favor CUDA, while models with $P < 0$ favor Triton.

The resulting picture is mixed rather than absolute. Several proprietary models, including GPT-5.5, Gemini 3.1 Pro, and Claude Opus 4.7, fall in the CUDA-preferred region, indicating that CUDA offers better runtime at lower or comparable token cost. Several open-weight models, including MiniMax-M2.5, DeepSeek-V4-Flash, and Devstral-2-123B, fall in the Triton-preferred region. Qwen3.5-397B-A17B sits close to the decision boundary, making it the most backend-agnostic open-weight option in our study.

Overall, the results suggest a practical rule of thumb: CUDA provides the highest performance ceiling for the strongest models, while Triton is often a better cost-performance choice for weaker or open-weight models.

5 Conclusion

We introduced **DataKernelBench** for evaluating LLM-generated GPU kernels for analytical query processing. Across ten LLMs on TPC-H SF10, GPT-5.5 with CUDA-`full` achieves **2.11**× over compiled TorchPlan with **100%** pass rate. The gains are query-dependent: plan inspection identifies kernel fusion, fused aggregation, and broader execution-strategy changes, while generic GPU engines remain competitive on more complex queries. To process data beyond one GPU's memory, a Dask-cuDF proof of concept executes **TPC-H SF100** in on-demand partitions distributed across four H100 GPUs; all 22 queries pass, with **2.54**× speedup over its partitioned TorchPlan baseline. These results establish LLM-generated kernels as a useful optimization path for selected recurring queries and DataKernelBench as a testbed for future work on LLM-driven database and GPU optimization.

6 Limitations

DataKernelBench is a first benchmark for this setting, and its current scope is intentionally controlled. First, the primary evaluation uses one fully instrumented setting: TPC-H at scale factor 10 on a single H100 GPU. Section 4.5 provides preliminary evidence at SF100 on four H100 GPUs, but broader evaluation across workloads, hardware generations, and data distributions is needed before drawing general conclusions about analytical processing settings. Second, our framework is centered on the TorchPlan execution model implemented in PyTorch, with Triton and CUDA as the main synthesis targets. Other acceleration stacks and programming interfaces, such as Numba, CuTe DSL, or emerging domain-specific GPU frameworks, are not yet included. Third, the primary pipeline assumes that the working set fits within available device memory. The proof of concept relaxes this assumption through partitioned multi-GPU execution, but does not systematically evaluate spilling, unified memory, or scaling behavior across cluster sizes. Finally, baseline TorchPlans are generated and then validated before benchmarking. This design is appropriate for studying kernel synthesis conditioned on a fixed tensor program, but it means that DataKernelBench does not evaluate the full path from arbitrary SQL input to a final optimized kernel.

Acknowledgments

This work is part of the ARMADA project (https://armada-dn.eu/) under the Marie Skłodowska-Curie Actions Doctoral Networks. The project is jointly funded by the Swiss State Secretariat for Education, Research and Innovation (SERI) under SBFI No. 24.00005 and by the European Union Horizon Europe research and innovation programme under Grant Agreement No. 101168951.

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the Swiss State Secretariat, the European Union, or the European Commission. Neither the Swiss State Secretariat, nor the European Union, nor the granting authorities can be held responsible for them.

References

MiniMax AI. 2026. Minimax-m2.5: A professional employee and sota model for agentic workflows. https://huggingface.co/MiniMaxAI/MiniMax-M2.5.

Anthropic. 2025. Introducing Claude Haiku 4.5. https://www.anthropic.com/news/claude-haiku-4-5. Accessed: 2026-05-26.

Anthropic. 2026a. Introducing Claude Opus 4.7. https://www.anthropic.com/news/claude-opus-4-7. Accessed: 2026-05-26.

Anthropic. 2026b. Introducing Claude Sonnet 4.6. https://www.anthropic.com/news/claude-sonnet-4-6. Accessed: 2026-05-26.

Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Dong He, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the tensor tea: How databases can leverage the machine learning ecosystem. *Proceedings of the VLDB Endowment*, 15(12):3598–3601.

Yuntao Bai, Saurabh Kadavath, Saurav Kundu, Amanda Askell, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2026. Kevin: Multi-turn RL for generating CUDA kernels. In *International Conference on Learning Representations*, volume 2026, pages 83418–83452.

Peter Boncz, Thomas Neumann, and Orri Erling. 2013. Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark. In *Revised Selected Papers of the 5th TPC Technology Conference on Performance Characterization and Benchmarking - Volume 8391*, page 61–76, Berlin, Heidelberg. Springer-Verlag.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, and 1 others. 2026. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*.

Google DeepMind. 2026. Gemini 3.1 Pro: A smarter model for your most complex tasks. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/. Accessed: 2026-05-26.

DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence.

Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2024. Text-to-sql empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment*, 17(5):1132–1145.

Gartner, Inc. 2025. Gartner says worldwide AI spending will total $1.5 trillion in 2025.

Dong He, Supun Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proceedings of the VLDB Endowment*, 15(11):2811–2825.

Zezhou Huang, Krystian Sakowski, Hans Lehnert, Wei Cui, Carlo Curino, Matteo Interlandi, Marius Dumitru, and Rathijit Sen. 2025. Gpu acceleration of sql analytics on compressed data. *Proceedings of the VLDB Endowment*, 19(3):320–333.

International Data Corporation (IDC). 2025. IDC: Artificial intelligence infrastructure spending to reach $758bn by 2029.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Kinetica. 2026. Kinetica. Accessed: 2026-03-13.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 489–504.

Robert Tjarko Lange, Qi Sun, Aaditya Prasad, Maxence Faldor, Yujin Tang, and David Ha. 2025. Towards robust agentic CUDA kernel benchmarking, verification, and optimization. *arXiv preprint arXiv:2509.14279*.

Jianling Li, Shangzhan Li, Zhenye Gao, Qi Shi, Yuxuan Li, Zefan Wang, Jiacheng Huang, WangHaojie WangHaojie, Jianrong Wang, Xu Han, and 1 others. 2025. Tritonbench: Benchmarking large language model capabilities for generating triton operators. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23053–23066.

Jinyang Li, Binyuan Hui, Ge Qu, Binyuan Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2024. Can llm already serve as a database interface? a big data benchmark for text-to-sql. *Advances in Neural Information Processing Systems (NeurIPS)*, 36.

Gang Liao, Hongsen Qin, Ying Wang, Alicia Golden, Michael Kuchnik, Yavuz Yetim, Jia Jiunn Ang, Chunli Fu, Yihan He, Samuel Hsia, and 1 others. 2025. Kernelevolve: Scaling agentic kernel coding for heterogeneous ai accelerators at meta. *arXiv preprint arXiv:2512.23236*.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. 2024. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*.

Jigao Luo, Nils Boeschen, Muhammad El-Hindi, and Carsten Binnig. 2025. Pystachio: Efficient distributed gpu query processing with pytorch over fast networks & fast storage. *arXiv preprint arXiv:2512.02862*.

Ryan Marcus. 2023. Learned query superoptimization. *arXiv preprint arXiv:2303.15308*.

Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment*, 12(11):1705–1718.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

OpenAI. 2025. gpt-oss-120b & gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

OpenAI. 2026. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/. Accessed: 2026-05-26.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. 2025. 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*, pages 47356–47415. PMLR.

Pandas. 2020. pandas-dev/pandas: Pandas.

Qwen Team. 2026a. Qwen3-coder-next technical report. Technical report. Accessed: 2026-03-13.

Qwen Team. 2026b. Qwen3.5: Towards native multimodal agents.

Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984.

RAPIDS. 2023. Rapids cudf: Gpu dataframe library. https://rapids.ai.

Abhinav Rastogi, Adam Yang, Albert Q Jiang, Alexander H Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Anmol Agarwal, Andy Ehrenberg, Andy Lo, and 1 others. 2025. Devstral: Fine-tuning language models for coding agent applications. *arXiv preprint arXiv:2509.25193*.

Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, pages 130–136.

SQream Technologies. 2026. SQream DB. Accessed: 2026-03-13.

Falcon-LLM Team. 2024. The falcon 3 family of open models, december 2024. *URL https://huggingface.co/blog/falcon3*.

Transaction Processing Council. 2018. Tpc benchmark h (tpc-h). https://www.tpc.org/tpch/.

Velox. 2022. Velox: A unified execution engine for data management systems. https://github.com/facebookincubator/velox.

Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. 2026. Bespoke olap: Synthesizing workload-specific one-size-fits-one database engines. *arXiv preprint arXiv:2603.02001*.

Zhongzhen Wen, Yinghui Zhang, Zhong Li, Zhongxin Liu, Linna Xie, and Tian Zhang. 2025. Multikernelbench: A multi-platform benchmark for kernel generation. *Preprint*, arXiv:2507.17773.

Jiin Woo, Shaowei Zhu, Allen Nie, Zhen Jia, Yida Wang, and Youngsuk Park. 2025. Tritonrl: Training llms to think and code triton without cheating. *arXiv preprint arXiv:2510.17891*.

Bowen Wu, Wei Cui, Carlo Curino, Matteo Interlandi, and Rathijit Sen. 2025. Terabyte-scale analytics in the blink of an eye. *Proceedings of the VLDB Endowment*, 19(2):141–155.

Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE.

Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei Gong, and Xiangyao Yu. 2026. Rethinking analytical processing in the gpu era. In *Conference on Innovative Data Systems Research (CIDR)*.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, and 1 others. 2025. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*.

Haitao Zhang, Ran Pang, Yuanyuan Zhu, Hao Zhang, Congli Gao, Ming Zhong, Jiawei Jiang, Tieyun Qian, and Jeffrey Xu Yu. 2025. Tqex: Tensor-based query engine enhanced by bridging the gap. *Proc. ACM Manag. Data*, 3(6).

tpch_q6_triton.py — Fused CUDA Kernel (LLM Optimized Core)

```python
def _query_core(
    l_shipdate_days, l_discount, l_quantity, l_extendedprice,
    shipdate_lo_days, shipdate_hi_days,
    discount_lo, discount_hi, quantity_hi,
) -> torch.Tensor:
    return _launch_q6_revenue(
        l_shipdate_days, l_discount, l_quantity, l_extendedprice,
        shipdate_lo_days, shipdate_hi_days,
        discount_lo, discount_hi, quantity_hi,
    )

def _launch_q6_revenue(...):
    revenue = _EXT.q6_revenue_launcher(   # torch.utils.cpp_extension.load
        l_shipdate_days, l_discount, l_quantity, l_extendedprice,
        shipdate_lo_days, shipdate_hi_days,
        discount_lo, discount_hi, quantity_hi,
    )
    return torch.tensor(revenue, dtype=torch.float64, device=l_extendedprice.device)

def run_query(...
```

```
__global__ void q6_revenue_kernel(...) {
    __shared__ double block_revenue[BLOCK_SIZE];

    const int64_t idx = blockIdx.x * blockDim.x + threadIdx.x;
    double my_revenue = 0.0;

    if (idx < n_elements) {
        // (1) Coalesced load: one thread per row
        const int32_t shipdate = shipdate_days_ptr[idx];

        // (2) Fused predicates in registers - no boolean mask tensor
        if (shipdate >= shipdate_lo_days && shipdate < shipdate_hi_days) {
            const double discount = discount_ptr[idx];
            if (discount >= discount_lo && discount <= discount_hi) {
                const double quantity = quantity_ptr[idx];
                if (quantity < quantity_hi) {
                    // (3) Conditional load of l_extendedprice
                    my_revenue = extendedprice_ptr[idx] * discount;
                }
            }
        }
    }

    // (4) Block reduce in shared memory
    block_revenue[threadIdx.x] = my_revenue;
    __syncthreads();
    for (int stride = blockDim.x >> 1; stride > 0; stride >>= 1) {
        if (threadIdx.x < stride)
            block_revenue[threadIdx.x] += block_revenue[threadIdx.x + stride];
        __syncthreads();
    }

    // (5) One atomicAdd per block
    if (threadIdx.x == 0)
        atomicAdd(out_ptr, block_revenue[0]);
}
```

Figure 5: TPC-H Q6 Example: Fused CUDA kernel. Like the Triton variant in Figure 2(C), `_query_core` delegates to a compiled launcher whose `__global__` kernel fuses filter, projection, and aggregation in one pass over `lineitem`. Nested branch predicates avoid materializing a boolean mask; `l_extendedprice` is loaded only for rows that pass all filters; partial sums are reduced in shared memory with one `atomicAdd` per block.

A Fused CUDA Kernel Example

Figure 5 shows the GPT-5.5 CUDA variant for the same TPC-H Q6 query as Figure 2. It uses the same TorchPlan interface: `run_query` handles cuDF I/O and parameters, while `_query_core` delegates to a compiled launcher and fused device kernel. We include this variant in the appendix because the main paper highlights the Triton core snippet in Figure 2(C).

tpch_q6_torchplan.py — TorchPlan with Dask-CuDF

```python
import torch
import cudf
import dask_cudf
import pandas as pd

def _query_core(
    l_shipday: torch.Tensor,
    l_extendedprice: torch.Tensor,
    l_discount: torch.Tensor,
    l_quantity: torch.Tensor,
    start_day: int,
    end_day: int,
    discount_low: float,
    discount_high: float,
    quantity_limit: float,
) -> torch.Tensor:
    keep = (
        (l_shipday >= start_day)
        & (l_shipday < end_day)
        & (l_discount >= discount_low)
        & (l_discount <= discount_high)
        & (l_quantity < quantity_limit)
    )
    return torch.sum(l_extendedprice[keep] * l_discount[keep])

def _process_partition(
    df_part: cudf.DataFrame,
    start_day: int,
    end_day: int,
    discount_low: float,
    discount_high: float,
    quantity_limit: float,
) -> float:
    l_shipday = (df_part["l_shipdate"].astype("int64") // 86400).astype("int32")
    result = _query_core(
        torch.as_tensor(l_shipday.to_cupy(), dtype=torch.int32),
        torch.as_tensor(df_part["l_extendedprice"].to_cupy(), dtype=torch.float64),
        torch.as_tensor(df_part["l_discount"].to_cupy(), dtype=torch.float64),
        torch.as_tensor(df_part["l_quantity"].to_cupy(), dtype=torch.float64),
        start_day,
        end_day,
        discount_low,
        discount_high,
        quantity_limit,
    )
    return float(result.item())

def run_query(
    tables: dict[str, dask_cudf.DataFrame],
    start_date: str = "1994-01-01",
    end_date: str = "1995-01-01",
    discount_low: float = 0.05,
    discount_high: float = 0.07,
    quantity_limit: float = 24.0,
) -> float:
    start_day = int(pd.to_datetime(start_date).timestamp() // 86400)
    end_day = int(pd.to_datetime(end_date).timestamp() // 86400)
    df_lineitem = tables["lineitem"][
        ["l_shipdate", "l_extendedprice", "l_discount", "l_quantity"]
    ]
    partials = df_lineitem.map_partitions(
        _process_partition,
        start_day,
        end_day,
        discount_low,
        discount_high,
        quantity_limit,
        meta=float,
    )
    return float(partials.sum().compute())
```

Figure 6: TPC-H Q6 Example: TorchPlan with Dask-cuDF. Like the eager-cuDF TorchPlan in Figure 2(B), `_query_core` implements the tensor hot path. The highlighted `_process_partition` converts each cuDF chunk to tensors and calls `_query_core`; `run_query` uses Dask-cuDF `map_partitions` to run those chunks on available GPU workers and sums the partial results.

B Partitioned Dask-cuDF TorchPlan Example

Figure 6 shows the partitioned TorchPlan for the same TPC-H Q6 query as Figure 2(B). The tensor hot path is unchanged; only table handling moves from eager cuDF to Dask-cuDF so that partitions can execute across GPUs. Section 4.5 uses this interface for SF100.

Figure 7: TPC-H Q11 Example: Multi-join TorchPlan. Complementing the single-table Q6 example, `run_query` handles cuDF table access, nation filtering, and the highlighted multi-table joins before converting joined columns to tensors; `_query_core` aggregates supply cost by part key, applies the HAVING threshold, and sorts. The `core` task preserves this boundary, whereas `full` may change the join strategy and how work is divided between the two functions.

C Multi-Join TorchPlan Example

Figure 7 shows the validated TorchPlan for TPC-H Q11. As in the Q6 CUDA example in Figure 5, the interface separates table handling from the tensor hot path: joins remain in `run_query`, while grouping, thresholding, and sorting run in `_query_core`.

Method	TorchPlan generator		Change
	Opus 4.7	GPT-5.5	
Baseline			
TorchPlan-`compile`	1.51 s	1.74 s	+15%
Optimized implementations			
GPT-5.5	0.71 s	0.89 s	+25%
Opus 4.7	1.00 s	0.91 s	-9%

Table 4: Total workload runtime (s) for TorchPlans from two generators. Optimized rows use CUDA-`full`; changes are approximate and relative to the Opus 4.7 TorchPlans. Lower is better.

D TorchPlan Generator Sensitivity

Claude Opus 4.7 was used for the main benchmark because it produced a complete validated set of **22/22 TorchPlans**; MiniMax M2.5 produced 20/22 under the same protocol. For the sensitivity analysis, GPT-5.5 also generated **22/22 valid TorchPlans**, but their compiled workload runtime was 1.74 s, approximately 15% slower than the 1.51 s runtime of the Opus 4.7 TorchPlans. Table 4 compares baseline and optimized runtimes across both sets.

Absolute runtimes change, but both optimized implementations remain faster than their corresponding baselines, and GPT-5.5 remains the strongest optimizer. Each optimizer is fastest on TorchPlans generated by the other strong model, suggesting that cross-model diversity may be useful. Overall, the main finding is not specific to TorchPlans generated by one model.

E Hardware and Software Environment

Table 5 reports the main software packages used for benchmark execution and kernel evaluation. Tables 6 and 7 report the GPU and CPU environments. The LLM-generated CUDA/Triton kernels and Sirius baseline use GPU acceleration, while DuckDB is used as a CPU-only database baseline.

F Repair Behavior

We analyze **880 generation trajectories**: 22 queries $\times$ 10 models $\times$ 2 frameworks $\times$ 2 optimization levels on H100 at SF10.

- **Repair effectiveness.** Overall, 77.5% eventually produce a correct kernel with at least 1.05$\times$ speedup. Cumulative success rises from 40% after round 1 to 57% after round 2

Package	Version
Python	3.13.11
pandas	2.3.3
cuDF	26.04.000
PyTorch	2.12.0+cu130
Triton	3.7.0
CUDA	13.0
DuckDB	1.5.2

Table 5: Software environment used for DataKernelBench experiments.

Field	Value
GPU	NVIDIA H100 SXM5
Architecture	Hopper (GH100)
SMs	132
CUDA cores	16,896
Tensor Cores	528, 4th gen
GPU memory	80 GB HBM3
Memory bandwidth	3.35 TB/s
L2 cache	50 MB
Peak FP64	34 TFLOPS
Peak FP32	67 TFLOPS
FP8 Tensor Core	Supported
TDP	Up to 700 W
Interconnect	NVLink 900 GB/s; PCIe Gen5 128 GB/s
Warp size	32

Table 6: GPU environment used for LLM-generated kernel evaluation and GPU-accelerated baselines.

Field	Value
CPU	Intel Xeon Platinum 8468
Architecture	Sapphire Rapids; 4th Gen Xeon Scalable
Sockets	2
Cores	96 total, 48 per socket
Logical CPUs	96, SMT off
NUMA nodes	2
Base frequency	2.10 GHz
Max turbo	3.80 GHz
L3 cache	210 MiB total
System memory	∼2.0 TB DDR5
TDP	350 W per socket
Socket	FCLGA4677, 2S platform
CPU database baseline	DuckDB 1.5.2, CPU-only

Table 7: CPU environment used for CPU-side execution and the DuckDB CPU-only baseline.

and 65% after round 3. Of the remaining trajectories, 17.5% never become correct, while 5% become correct but remain below the speedup threshold.

- **Failure categories.** Among round-1 failures, approximately 38% fail to compile or import, 52% crash or time out during execution, and 9% return incorrect output, primarily wrong values or row counts; column mismatches account for less than 1%.

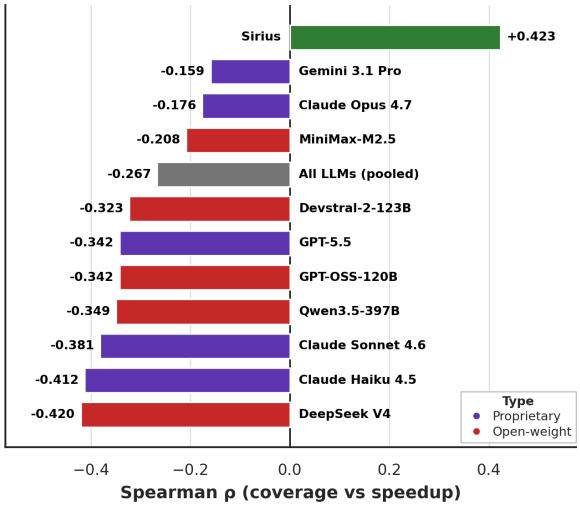

Figure 8: Spearman correlation between choke-point coverage and per-query speedup. Coverage counts the number of distinct choke-point categories assigned to each TPC-H query. All ten LLMs show negative correlation, while Sirius shows positive correlation

- **Backend differences.** Approximately 74% of CUDA failures are compilation or import errors, whereas 79% of Triton failures are execution errors.

- **Query difficulty.** Q13 is the hardest query, with 13/40 successful trajectories. Q20, Q3, and Q1 are also difficult, while Q6 and Q19 succeed in all 40 trajectories.

- **Prompt improvements.** Recurring failures led us to pin runtime versions and add guidance on separate kernel source strings and epoch-day date representation, reducing repair rounds without materially changing final speedup.

G Choke-Point Coverage Analysis

To better understand when LLM-generated kernels outperform a general-purpose GPU database baseline, we define choke-point coverage as the number of distinct choke-point categories (Boncz et al., 2013) explicitly associated with each TPC-H query. We count only query-category assignments named in the taxonomy, making the measure conservative: queries may involve additional relational patterns that are not counted in our annotation.

For each system, averaged across 4 framework-optimization level combinations, we compute the

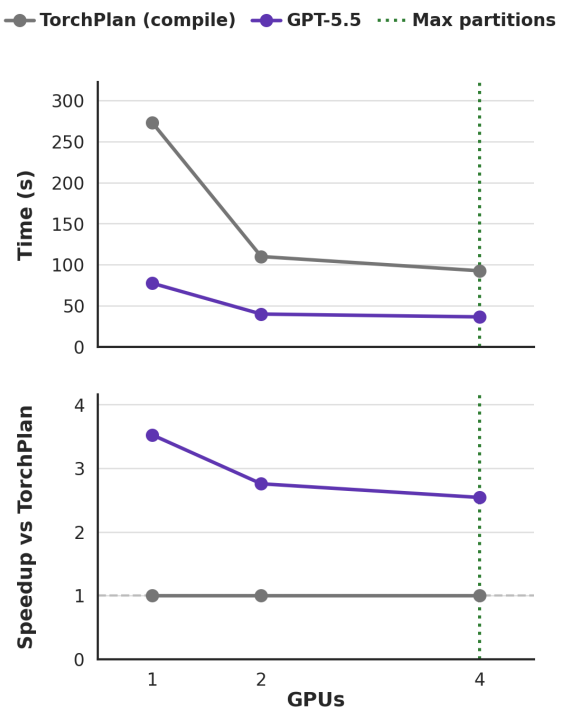

Figure 9: Multi-GPU scaling of GPT-5.5 CUDA-`full` kernel plans versus TorchPlan-`compile` on TPC-H SF100 (H100, Dask-cuDF, 10 GB chunks). The green dotted line marks the widest table partition count (`lineitem` = 4). **Top:** runtime over all 22 queries. **Bottom:** overall speedup versus TorchPlan. The CUDA kernel plan advantage over TorchPlan is largest on one GPU ($3.53\times$) and is $2.54\times$ at 4 GPUs.

Spearman rank correlation between the 22 per-query coverage values and the corresponding 22 per-query speedups. Figure 8 shows that all ten LLMs have negative correlation between coverage and speedup, while Sirius has positive correlation. When pooling all LLM-query pairs, the trend remains negative with $\rho = -0.267$ ($n = 220$). We also find that the head-to-head LLM/Sirius speedup ratio decreases with coverage for all ten LLMs. These results support the complementary interpretation in Section 4.4: current LLM-based specialization is most effective for narrower query patterns, while Sirius becomes relatively stronger on queries that combine more relational choke points.

H Multi-GPU Scaling

Figure 9 varies the GPU count from 1 to 4 for the SF100 setting in Section 4.5. Dask-cuDF represents each table as partitions and, through `map_partitions`, runs independent partitions concurrently on the available GPU workers, queuing any remaining partitions until a worker is free. Under this layout `lineitem` has four partitions and every other table has one, so four is the maximum number of partitions that can run at once. We therefore stop at four GPUs. As in Section 4.1, queries that fail or do not reach $1.05\times$ are credited at the TorchPlan-`compile` runtime. Runtime over all 22 queries falls from 1 to 4 GPUs for both the CUDA kernel plans (77.5 s to 36.44 s) and TorchPlan-`compile` (273 s to 92.6 s).

At higher scale factors, more partitions would be available, so additional GPUs could run more partitions in parallel. A possible further improvement is to rewrite Dask-cuDF multi-GPU orchestration with LLM-generated native PyTorch code.