DataKernelBench: Can LLMs Optimize Database Queries on GPUs?

Anonymous ACL submission

Abstract

GPUs are increasingly present in analytical data systems, yet high performance for relational workloads often still depends on hand-written kernels beyond what generic ML compiler stacks provide. While recent LLMs have shown strong performance on GPU kernel synthesis for machine learning operators, existing benchmarks do not test whether they can optimize database-style queries, which are memory-bound, irregular, and composed of multiple relational operators. We introduce **DataKernelBench**, a benchmark for evaluating LLM-generated Triton and CUDA kernels for analytical query processing. Each SQL query is translated into **TorchPlan**, a validated PyTorch tensor program that separates table handling from the tensor-intensive execution path. LLMs then specialize either the hot path or the full query using execution-guided repair. We evaluate pass rate, speedup over `torch.compile`, and generation cost. On TPC-H scale factor 10 on an H100 GPU, GPT-5.5 with full-query CUDA specialization achieves **2.11**× speedup over `torch.compile` with **100%** pass rate, including **11.2**× on Q14. We further find that full-query specialization disproportionately helps stronger LLMs, that workload context in the prompt matters more than GPU-hardware context, and and stronger models tend to benefit more from CUDA than Triton.

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.

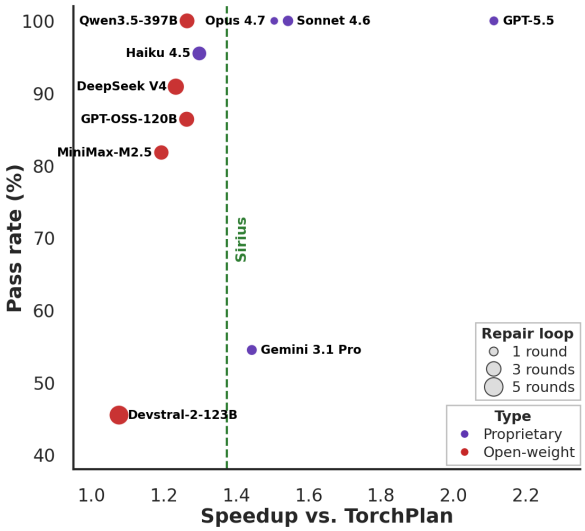

Figure 1: DataKernelBench tests LLMs on GPU kernel generation for database query optimization, where the strongest frontier models achieve 100% pass rate and substantial speedups over the TorchPlan baseline on TPC-H SF10 with NVIDAI H100. See Table 1 and §4.2.

This combination makes GPU acceleration an increasingly attractive direction for analytical data processing (Yogatama et al., 2025a), 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 ML frameworks such as PyTorch. Recent work shows this approach can scale to 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 modern code LLMs synthesize specialized GPU kernels that outperform generic compilation?

This question matters because generic ML compiler stacks can still leave substantial performance

on the table for analytical workloads. Although `torch.compile` is highly effective for regular, compute-bound neural operators, analytical queries often involve irregular memory access, complex predicates, heterogeneous 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) suggests a promising path toward reducing this engineering burden. Prior work shows LLMs can generate specialized CUDA and Triton kernels for hardware acceleration (Woo et al., 2025; Dai et al., 2026; Liao et al., 2025), and recent benchmarks such as KernelBench, TritonBench, and MultiKernelBench evaluate this capability on ML operators (Ouyang et al., 2025; Li et al., 2025; Wen et al., 2025). However, these benchmarks focus on regular tensor computations with predictable structure. Analytical query processing is different: it is often memory-bound, combines multiple relational operators, and requires strict correctness under irregular 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**, a benchmark 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 against DuckDB (Raasveldt and Mühleisen, 2019) before kernel generation. The benchmark then asks frontier and open-weight LLMs to inject optimized Triton or CUDA kernels under a multi-round execution-guided repair loop.

Our primary comparison point 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., 2025b) 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.

With this framework, we evaluate ten frontier and open-weight LLMs across CUDA and Triton backends and both optimization levels (§4). Gains are highly query-dependent, and the best LLM optimized kernels complement rather than uniformly replace a generic GPU database. Overall, frontier LLMs already produce correct kernels with meaningful speedups on a standard workload.

This work makes three contributions. First, we introduce, to our knowledge, the first systematic benchmark for LLM-generated GPU kernels on analytical query workloads rather than ML operators. Second, we provide a validated SQL-to-TorchPlan pipeline and the resulting modular Python TorchPlan programs. They exposes an LLM-friendly optimization target, and serves as a pluggable reference implementation for kernel synthesis. Third, using DataKernelBench we identify four empirical patterns of current code LLMs on this task: full-query specialization helps stronger models more, workload context in the prompt matters more than hardware context, backend preference is model-dependent with stronger models more often benefiting from CUDA, and ultimately frontier LLMs in aggregate outperform a strong general-purpose GPU database baseline.

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., 2025b), 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. Recent work shows that this approach can scale to terabyte-scale multi-GPU analytics (Wu et al., 2025) and can be extended toward distributed, storage-resident OLAP over PyTorch-based execution engines (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-oriented LLMs has led to rapid 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 differs in important ways: it is often memory-bound, combines multiple relational operators, and requires strict correctness under irregular control flow. 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.

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. Although Q6 is a simple single-table example, the benchmark covers all 22 TPC-H queries, having 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. We share the validated TorchPlans together with GPT-5.5 `full`-level CUDA and Triton implementations in the supplementary material.

3.1 Benchmark Workload and Setting

Our current release focuses on a single fully instrumented benchmark track: **TPC-H** (Transaction Processing Council, 2018) at **scale factor 10**. TPC-H contains 22 standardized analytical queries over

tpch_q6.sql (A) SQL (Analytical Intent)

```sql
-- TPC-H Q6:compute discounted revenue on lineitem table
SELECT SUM(l_extendedprice * l_discount) AS revenue
FROM lineitem
WHERE l_shipdate >= DATE '1994-01-01'
  AND l_shipdate < DATE '1995-01-01'
  AND l_discount BETWEEN 0.05 AND 0.07
  AND l_quantity < 24
```

tpch_q6_torchplan.py (B) TorchPlan (LLM Generated Baseline)

```python
def _query_core(
  l_shipdate_days: torch.Tensor,
  l_discount: torch.Tensor,
  l_quantity: torch.Tensor,
  l_extendedprice: torch.Tensor,
  shipdate_lo_days: int,
  shipdate_hi_days: int,
  discount_lo: float,
  discount_hi: float,
  quantity_hi: float,
) -> torch.Tensor:
  # Tensor hot path — LLM replaces this in kernel synthesis
  mask = (
    (l_shipdate_days >= shipdate_lo_days)
    & (l_shipdate_days < shipdate_hi_days)
    & (l_discount >= discount_lo)
    & (l_discount <= discount_hi)
    & (l_quantity < quantity_hi)
  )
  return torch.sum(l_extendedprice[mask] * l_discount[mask])

def run_query(
  tables: dict[str, cudf.DataFrame],
  shipdate_lo: str = "1994-01-01",
  shipdate_hi: str = "1995-01-01",
  discount_lo: float = 0.05,
  discount_hi: float = 0.07,
  quantity_hi: float = 24.0,
) -> float:
  # cuDF I/O + SQL literals as keyword arguments with defaults
  df = tables["lineitem"]
  shipdate_lo_days = int(pd.to_datetime(shipdate_lo).timestamp() // 86400)
  shipdate_hi_days = int(pd.to_datetime(shipdate_hi).timestamp() // 86400)

  l_shipdate_days = torch.as_tensor(
    ((df["l_shipdate"].astype("int64") // 86400).astype("int32")).to_cupy()
  )
  l_discount      = torch.as_tensor(df["l_discount"].to_cupy(), dtype=torch.float64)
  l_quantity      = torch.as_tensor(df["l_quantity"].to_cupy(), dtype=torch.float64)
  l_extendedprice = torch.as_tensor(df["l_extendedprice"].to_cupy(), dtype=torch.float64)

  return float(_query_core(
    l_shipdate_days, l_discount, l_quantity, l_extendedprice,
    shipdate_lo_days, shipdate_hi_days,
    discount_lo, discount_hi, quantity_hi,
  ).item())
```

tpch_q6_triton.py (C) Fused Triton 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_triton(
        l_extendedprice, l_discount, l_quantity, l_shipdate_days,
        shipdate_lo_days, shipdate_hi_days,
        discount_lo, discount_hi, quantity_hi,
    )

def _launch_q6_triton(price, discount, quantity, shipdate_days,
                      date_lo, date_hi, disc_lo, disc_hi, qty_hi):
    out = torch.zeros(1, dtype=torch.float64, device=price.device)
    grid = (triton.cdiv(price.numel(), BLOCK_SIZE),)
    _q6_fused_filter_project_sum_kernel[grid](
        price, discount, quantity, shipdate_days,
        date_lo, date_hi, thresholds, out, price.numel(),
        BLOCK_SIZE=BLOCK_SIZE,
    )
    return out.squeeze(0)

@triton.jit
def _q6_fused_filter_project_sum_kernel(...):
    # (1) Coalesced column loads
    shipdate = tl.load(shipdate_days_ptr + offs, mask=in_bounds, other=date_hi)
    discount = tl.load(discount_ptr + offs, mask=in_bounds, other=0.0)
    quantity = tl.load(quantity_ptr + offs, mask=in_bounds, other=qty_hi)

    # (2) Fused predicates in registers — no boolean mask tensor
    keep = (
        in_bounds
        & (shipdate >= date_lo) & (shipdate < date_hi)
        & (discount >= disc_lo) & (discount <= disc_hi)
        & (quantity < qty_hi)
    )

    # (3) Conditional load  (4) project  (5) fused reduce
    price   = tl.load(price_ptr + offs, mask=keep, other=0.0)
    contrib = tl.where(keep, price * discount, 0.0)
    tl.atomic_add(out_ptr, tl.sum(contrib), sem="relaxed")

def run_query(...
```

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 in the appendix.

8 relational tables. We generate benchmark artifacts using DuckDB's `tpch` extension, including table CSVs, SQL query files, 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 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 compute-heavy region is isolated in a form suitable 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.

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. These validated TorchPlans serve as both the correctness reference for synthesized kernels and the baseline execution path from which speedup is measured.

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`: `run_query` is fixed, and the model may replace only `_query_core` and private helper functions;
- `full`: the model may rewrite the internal query implementation while preserving the `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. We share all the prompts in the supplementary material.

3.5 Correctness and Speedup Criteria

For each candidate, we compare the output of `run_query` against the validated TorchPlan on the fixed scale factor 10 tables. We report mismatches hierarchically: output shape, column names and order, row count, and then cell-level differences with sampled offending values. Non-floating columns require exact equality. 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` latency. Let T_{base} denote the median latency of the validated TorchPlan when `_query_core` is wrapped with `torch.compile`, and let T_{cand} denote the candidate median latency after warmup. We define

$$\text{speedup} = T_{\text{base}}/T_{\text{cand}}.$$

A candidate passes inside the generation loop only if it is functionally correct and achieves speedup $\geq s_{\min}$. This reflects the intended deployment setting: when an optimization attempt fails to improve performance, the system would retain the compiled TorchPlan baseline.

3.6 Execution-Guided Generation

For each query, model, framework, and optimization level, we run a multi-round execution-guided repair loop with at most $R_{\max}$ rounds. Round 1 receives the composed prompt from §3.4. Later rounds append feedback from the previous attempt, including validation failures, output mismatches, tracebacks, and measured speedup. We stop as soon as one candidate passed correctness test and satisfies speedup $\geq s_{\min}$. Correctness checks and warmup runs are terminated after $\max(T_{\min}, M_{\text{warmup}} \cdot T_{\text{base}})$, while each timed run is terminated after $M_{\text{timed}} \cdot T_{\text{base}}$. These limits prevent invalid or hung kernels from blocking evaluation. 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.

Model	Best Config	Pass Rate%	↑ Speedup vs. TC Overall	≥1.05×	≥1.20×	≥2×	↓ Runtime Overall (s)	N_{save1h}	↓ Mean Generation Cost Rounds	In	Out	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 vs. TC** 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×, 1.20×, and 2× speedups. N_{save1h} represents the number of repetitions required to save 1 hour of execution time relative to TC. **In/Out** denotes token usage in thousands; **USD** is the mean LLM generation cost per query. Arrows indicate higher (↑) or lower (↓) values are better. The best results are **bolded** and second-best are underlined.

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). 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.

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 B.

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 also 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 budget on unsuccessful repairs. For example, GPT-5.5

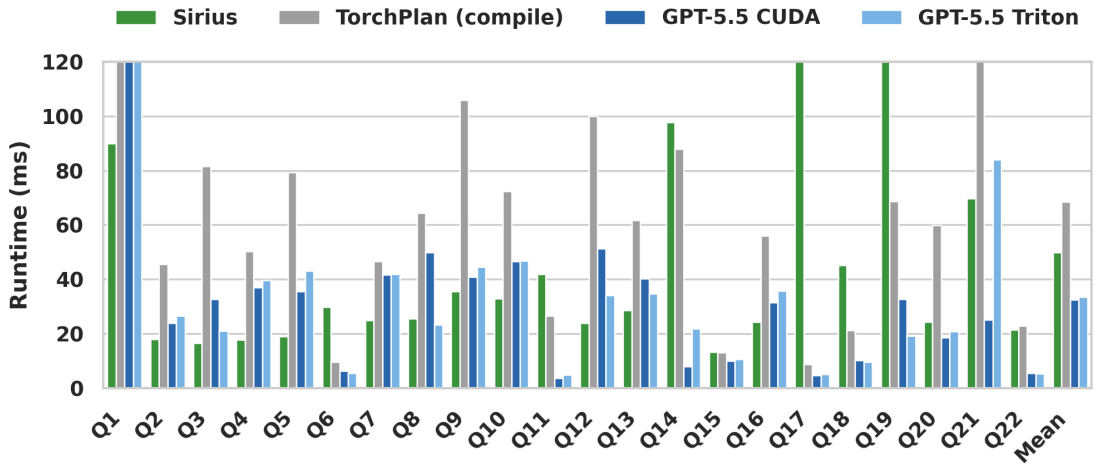

Figure 3: Per-query runtime comparison: GPT-5.5 CUDA/Triton at Full level vs. baselines. Y-axis capped at 120ms.

	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.

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.

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 Query-Level Analysis

Figure 3 shows per-query runtime for TorchPlan-`compile`, Sirius, and the best GPT-5.5 `full`-level configurations in CUDA and Triton. The main pattern is strong non-uniformity across queries. Sirius is fastest on many queries and GPT-5.5-generated kernels dominate on a different subset, including the most important wins in the benchmark.

The clearest example is Q14, where GPT-5.5 achieves the largest single-query gain and substantially outperforms both TorchPlan-`compile` and Sirius. Similar advantages appear on queries such as Q17 and Q19. These results support a complementary interpretation rather than a universal one: bespoke LLM-synthesized kernels are not a replacement for generic GPU databases on every query, but they can deliver very large wins on selected analytical pipelines, large enough to change the benchmark-wide ranking. Appendix C further analyze this using the TPC-H choke-point taxonomy (Boncz et al., 2013). All ten LLMs show negative correlation between choke-point coverage and per-query speedup, while Sirius shows a positive correlation, suggesting that LLM-generated kernels are strongest on narrower query patterns whereas Sirius becomes more competitive as queries combine more relational choke points.

4.4 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

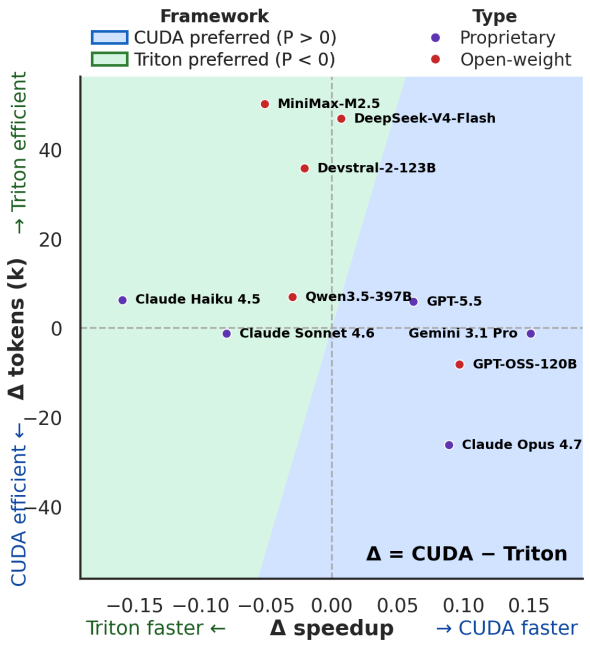

Figure 4: CUDA vs. Triton preference per model. Δspeedup and Δtokens (k) measure the runtime and token usage change between frameworks. The boundary separates CUDA-preferred ($P > 0$) from Triton-preferred ($P < 0$) regions at default $\lambda = 10$.

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.

4.5 Prompt Ablation

Table 3 reports prompt ablations for GPT-5.5 at the `full` level. 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

4.6 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**, a benchmark for evaluating whether LLMs can synthesize efficient GPU kernels for analytical query processing. DataKernelBench converts SQL queries into validated **TorchPlan** programs and then measures how well frontier and open-weight LLMs inject optimized Triton or CUDA kernels under correctness and runtime constraints. On TPC-H at scale factor 10 on an H100 GPU, GPT-5.5 with `full`-level CUDA specialization achieves **2.11**$\times$ speedup over TorchPlan-`compile` with **100%** pass rate, including **11.2**$\times$ on Q14. Our experiments further show that gains are highly query-dependent, that stronger models benefit substantially more from full-query specialization than weaker ones, and that workload context in the prompt matters more than hardware context. These results suggest that LLM-driven kernel synthesis is a promising optimization path for selected analytical workloads, and that DataKernelBench can serve as a useful testbed for future work at the intersection of LLMs, database systems, and GPU programming.

6 Limitations

DataKernelBench is a first benchmark for this setting, and its current scope is intentionally controlled. First, the present study evaluates one fully instrumented track: TPC-H at scale factor 10 on a single H100 GPU. This choice makes the comparisons reproducible and keeps the benchmark focused, but broader evaluation across additional workloads, scale factors, hardware generations, and data distributions is needed before drawing general conclusions about all 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 current pipeline assumes that the working set fits within available device memory. Larger-scale execution with spilling, streaming, unified memory, or multi-GPU partitioning is an important direction, but requires additional systems support beyond the present benchmark. 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.

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 1 others. 2022. Share the tensor tea: how databases can leverage the machine learning ecosystem. *arXiv preprint arXiv:2209.04579*.

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

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. *arXiv preprint arXiv:2203.01877*.

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

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.

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*.

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? *arXiv preprint arXiv:2502.10517*.

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*.

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

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. *arXiv preprint arXiv:2506.09226*.

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, and 1 others. 2025a. Rethinking analytical processing in the gpu era. *arXiv preprint arXiv:2508.04701*.

Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, and 1 others. 2025b. Rethinking analytical processing in the gpu era. *arXiv preprint arXiv:2508.04701*.

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*.

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).

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 4: Software environment used for DataKernel-Bench 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 5: 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 6: CPU environment used for CPU-side execution and the DuckDB CPU-only baseline.

B Hardware and Software Environment

Table 4 reports the main software packages used for benchmark execution and kernel evaluation. Tables 5 and 6 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.

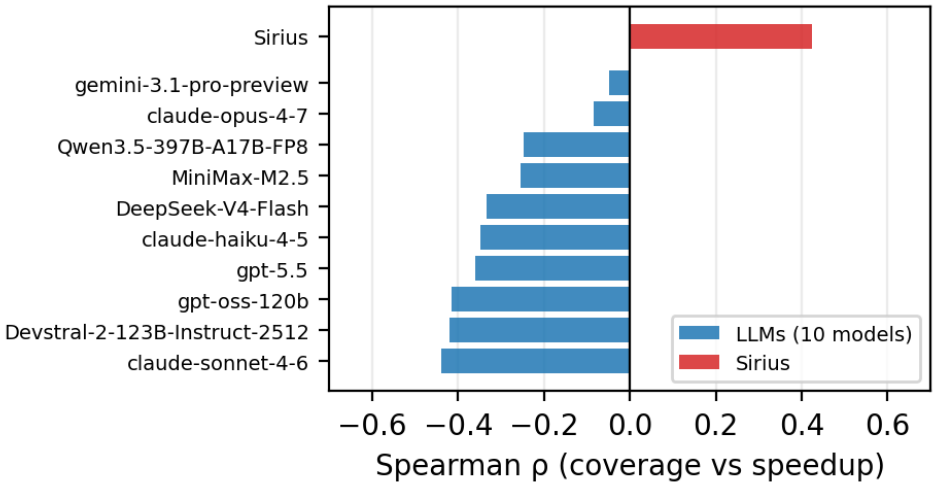

Figure 6: Spearman correlation between choke-point coverage and per-query speedup. Coverage counts the number of distinct choke-point categories assigned to each TPC-H query. All ten LLMs show negative correlation, while Sirius shows positive correlation

C 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, we compute the Spearman rank correlation between the 22 per-query coverage values and the corresponding 22 per-query speedups. Figure 6 shows that all ten LLMs, in their best framework-level configurations, have negative correlation between coverage and speedup, while Sirius has positive correlation. When pooling all LLM-query pairs, the trend remains negative with $\rho = -0.25$ ($p = 0.0002$, $n = 220$). 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 4.3: current LLM-based specialization is most effective for narrower query patterns, while Sirius becomes relatively stronger on queries that combine more relational choke points