Automated Operator Fusion for GPU-Accelerated SQL: Generation and Evolutionary Refinement

Corey Lammie[†]
IBM Research Europe
Rüschlikon, Switzerland
corey.lammie@ibm.com

Yotam Perlitz
IBM Research Europe
Rüschlikon, Switzerland
y.perlitz@ibm.com

Andrea Giovannini
IBM Research Europe
Rüschlikon, Switzerland
agv@zurich.ibm.com

Abdel Labbi
IBM Research Europe
Rüschlikon, Switzerland
abl@zurich.ibm.com

ABSTRACT

GPU-accelerated SQL engines improve throughput and cost efficiency over CPU-based engines by lowering relational operators to GPU kernels. However, each kernel invocation pays a launch cost and forces intermediate data through a global-memory round-trip. Kernel fusion reclaims these costs, but two questions remain central for data-processing workloads: at what *granularity* should fusion be applied, and how can fused kernels be *produced* automatically rather than hand-written? This materializes in a design space spanning the *specificity* of the fused kernels, the *automation/runtime costs* for their generation, and their *performance*. We study this design space in the context of the cuDF backend of Velox, targeting the middle ground between per-operator hand-written kernels and bespoke implementations synthesized for an entire workload or execution engine. Our approach first lowers query plans into structural fusion dispatch rules and fused-operator scaffolds, then uses an iterative LLM loop to fill each scaffold with a functionally correct fused Triton kernel — a *seed* — across four code-generation models. These seeds are subsequently optimized with evolutionary search through SkyDiscover. Across TPC-H-derived pipeline windows, we find that strong code models can often generate buildable and equivalent fused Triton kernels, and that these kernels can improve end-to-end runtime when deployed opportunistically with fallback. We also show, on selected representative seeds, that evolutionary refinement can further improve performance.

VLDB Workshop Reference Format:
Corey Lammie, Yotam Perlitz, Andrea Giovannini, and Abdel Labbi. Automated Operator Fusion for GPU-Accelerated SQL: Generation and Evolutionary Refinement. VLDB 2026 Workshop: 17th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS26).

1 INTRODUCTION

High-performance accelerated SQL execution is typically organized as pipelines of relational operators. This abstraction is essential for optimization and scheduling, but operator boundaries are not always the right execution boundaries for modern accelerators. For GPUs in particular, kernel selection, fusion, and composition can have a first-order effect on performance.

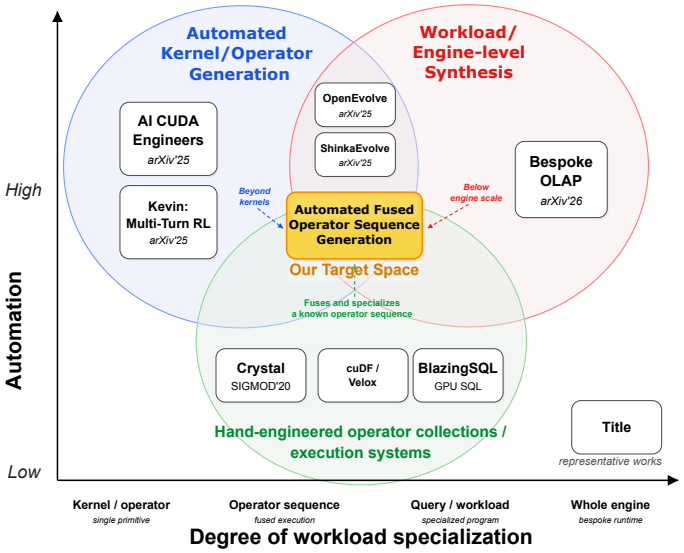

Figure 1: Existing automated kernel-generation systems primarily target individual GPU kernels or operators, while hand-engineered data-processing systems rely on substantial execution logic, operator implementations, and fusion strategies. Recent evolutionary and LLM-based approaches span both kernel-level automation and workload- or engine-scale synthesis. Our work occupies the middle ground: it automatically generates fused, specialized implementations for known sequences of data-processing operators, targeting pipeline-granularity execution without requiring whole-engine synthesis.

In practice, many GPU-based SQL engines preserve the streaming operator structure of the execution engine while lowering each operator boundary to separate library calls or kernel invocations over columnar batches. This design is practical: it mirrors the CPU execution stack, supports batching and multi-tenancy, allows GPU and CPU operators to coexist, and maximizes reuse of libraries such as RAPIDS cuDF [44]. Systems such as Velox [40] and Sirius [37] illustrate the appeal of this style of GPU-accelerated analytics. Its drawback is not that execution stops being pipelined, but that the physical GPU work remains segmented at operator boundaries. Each boundary can introduce additional kernel launches, synchronization, scheduling overhead, or intermediate column materialization.

[†]Corresponding author.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

At the other end of the design space are bespoke, workload-specific engines, e.g., Bespoke OLAP [56]. These systems generate or hand-specialize complete workloads. Concretely, this means specializing data layout, scan and aggregation kernels, and query-specific code paths together, so that each workload is served by a dedicated engine rather than by a shared operator stack. This direction can deliver excellent performance, but it also changes the engineering problem: the generated artifact is larger, less reusable, and more expensive to validate. A workload revision, a plan change, or a new operator pattern can require substantial regeneration or manual effort.

This paper asks whether an existing data-processing engine can retain its operator infrastructure and fallback path while automatically producing fused GPU kernels for recurring operator windows (Fig. 1). We do not aim to replace a database optimizer, synthesize a new database system for each workload, replace each operator independently, or prove that LLM-generated kernels dominate hand-written ones. Instead, we target bounded operator sequences: each candidate window is automatically generated, validated against the unfused reference path, and opportunistically deployed with fallback, while unmatched regions continue to execute through an existing backend.

Operator fusion makes this middle ground attractive. A fused kernel replaces a sequence of operators with a single GPU kernel, amortizing launch overhead and keeping intermediate values in registers or shared memory rather than materializing them in global-memory. Fusion is well established in deep-learning compilers [7, 15, 45] and appears in hand-tuned GPU data systems [46], however remains largely unexplored for data-processing engines.

This brings the design space into focus. A short filter-project-aggregate window is often an attractive fusion candidate, but fusing across joins, repartitioning, or large grouped aggregations can increase register pressure, reduce occupancy, and make the implementation too specific to reuse. At the same time, writing high-quality fused kernels by hand for every recurring pipeline does not scale, and hand-written kernels are brittle when plans change.

We instantiate this design point in a system built on the GPU-accelerated cuDF backend of Velox [40, 54]. Velox provides a production-oriented execution substrate with a clear operator abstraction and an existing per-operator GPU path. Our system preserves that path as fallback, but extends the backend with a fused execution path written in Triton [53], a Python-embedded GPU DSL, ahead of it. When a contiguous operator window matches a registered fusion dispatch rule, Velox replaces the window with a fused GPU operator backed by a Triton kernel; operators not covered by a rule continue through the cuDF backend. Because Velox is used by multiple query engines, our extensions are inherited by the systems that build on it, including Presto [50], Spark via Gluten [47], and Axiom [29].

This design point is realized in three stages. First, we lower query plans into structural fusion dispatch rules and C++ fused-operator scaffolds. Each rule describes a contiguous operator window using predicates over plan-node properties, and each scaffold registers the corresponding fused operator. Second, a code-generation model fills the scaffold with a Triton implementation, producing a *seed* that must compile and pass an equivalence test against the unfused Velox pipeline. Third, seeds enter a refinement loop that mutates only the Triton kernel body while preserving the same build and equivalence gates. Our specific contributions are as follows:

- We formulate pipeline-granularity fused kernel generation for GPU SQL execution as a distinct design point and represent fusion opportunities explicitly through rule-based operator windows in Velox;
- We realize this design point through a Triton-backed fused execution path using the cuDF backend, maximizing reuse of the existing infrastructure;
- We present a gated synthesis-and-refinement workflow that turns plan-derived fused-operator scaffolds into validated Triton kernels: code models generate seed kernels, build and equivalence checks filter them, and evolutionary refinement refines seeds under the same correctness gates;
- We evaluate four code-generation models across 21 TPC-H queries and characterize seed-kernel correctness, convergence, and speedup versus the per-operator cuDF baseline;
- For four hand-selected seeds, we present a preliminary feasibility study showing that evolutionary search can improve selected seeds, and compare two representative targets against independently hand-tuned Triton implementations.

2 BACKGROUND AND MOTIVATION

2.1 Velox Operator Pipelines

Velox represents a query fragment as a sequence of physical operators executed by one or more *Drivers*. Each plan node is instantiated as an `Operator` subclass, such as `FilterProject`, `HashJoin`, or `PartialAggregation`, and operators exchange Arrow-compatible columnar batches called Velox *vectors*. Fig. 2 illustrates this execution model.

As shown in Fig. 2a, the existing GPU backend replaces supported Velox operators with cuDF-backed counterparts. Each `CudfOperator` lifts its input vectors into device-resident cuDF columns and invokes one or more precompiled GPU kernels. Unsupported operators, data types, or expressions remain on the ordinary Velox execution path. This design is attractive because it preserves the Velox planner, CPU execution path, and operator abstraction. However, this design also places GPU execution at the per-operator extreme of the specificity axis in Fig. 1. Every operator boundary becomes a kernel-launch boundary, and every intermediate column produced by one GPU operator must be materialized to global memory before the next operator can consume it. Consequently, short operators such as projections, filters, and low-cardinality aggregations may spend a significant fraction of their runtime on launch overhead and intermediate memory traffic rather than useful computation. Moreover, because each operator is optimized in isolation, the GPU backend cannot exploit locality across adjacent operators in the same pipeline.

Our approach keeps the existing Velox operator front-end and retains the per-operator cuDF path as a fallback, but adds a pipeline-granularity execution path ahead of it. As shown in Fig. 2c, when a contiguous window of `CudfOperators` matches a registered fusion dispatch rule, we replace that window with a single fused GPU operator backed by one generated Triton kernel. This lets us target

the per-pipeline band of Fig. 1 without rewriting the IO streaming and caching logic, query planner, or pipeline synchronization management.

2.2 Operator Fusion Fundamentals

Operator fusion replaces a sequence of operators $o_1 \rightarrow o_2 \rightarrow \cdots \rightarrow o_k$ with a single kernel $K_{1..k}$ that directly produces the final output of the sequence [20, 31]. For GPU execution, this has two primary benefits. First, the sequence pays one kernel-launch latency instead of k launch latencies. Second, intermediate values can remain in registers, shared memory, or other on-chip storage rather than being written to and reread from global memory after every operator.

The natural unit of fusion in Velox is the pipeline. As shown in Fig. 2b, Velox plans are split at blocking boundaries such as exchanges, `localPartition()`, and hash-join build phases. Within a pipeline, operators are non-blocking: a batch can flow from one operator to the next without requiring the entire upstream result to be materialized. Such regions are therefore well suited to fusion, since the fused kernel can preserve the original streaming dataflow while eliminating intermediate materialization. Pipeline boundaries, in contrast, already impose a synchronization point on the CUDA stream before the next pipeline can consume the materialized or repartitioned output.

Blocking boundaries impose a stricter constraint. Across such a boundary, the upstream side must finish producing a materialized buffer before the downstream side can proceed. Fusion across the boundary is legal only if the fused kernel also implements the required materialization semantics. Such fused kernels can remove additional launches and memory traffic, but they also couple more of the plan into a single specialized kernel and may increase compilation cost, register pressure, and scheduling complexity. Accordingly, this paper focuses on the simpler and more broadly applicable case shown in Fig. 2c, fusing *contiguous, in-pipeline windows* of `CudfOperators`.

Fusion is not always beneficial. A larger fused kernel must hold more live state at once, which can increase register pressure and reduce occupancy. Moreover, fusion can reduce reuse: a kernel specialized for the sequence $o_1 \rightarrow o_2$ cannot necessarily be reused for a query that contains only o_1 or for a similar sequence with different expressions, types, or layout assumptions. These trade-offs motivate the central design problem addressed throughout the rest of the paper.

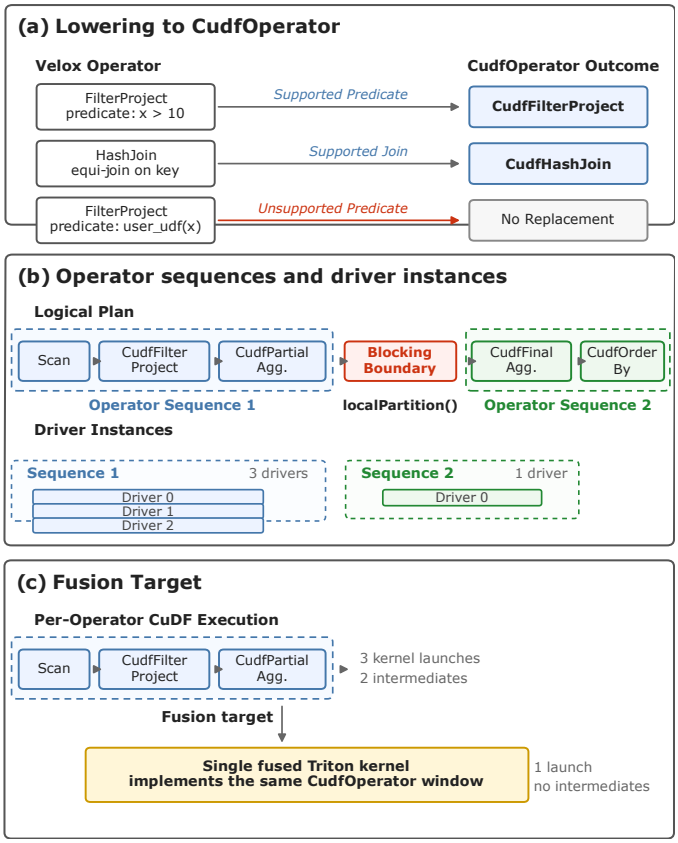

Figure 2: Velox execution model and fusion target. (a) Supported Velox operators are lowered to `CudfOperators`. (b) Query plans are split into operator sequences at blocking boundaries such as `localPartition()`, with each sequence executed by one or more drivers. (c) Rather than launching one cuDF kernel per `CudfOperator` and materializing intermediates, we fuse contiguous `CudfOperator` windows into a single Triton kernel.

3 SYSTEM SUPPORT FOR AUTOMATED FUSION

This section describes the system mechanisms that enable automated fusion in Velox. Rather than hard-coding fusion opportunities into the query engine, we separate fusion into three layers: ① a declarative rule layer that identifies fusible operator windows, ② a registry layer that binds matched rules to fused-operator implementations, and ③ a kernel-execution layer that allows fused operators to target specific sequences of operators.

3.1 Declarative Fusion Dispatch Rules

The fusion system is driven by JSON configuration files that describe operator patterns and the structural predicates those patterns must satisfy. Rule activation, operator-window matching, and predicate validation are therefore specified outside the matcher itself. Trying a new fusion opportunity does not require modifying the matching engine; it requires enabling a new rule and providing the corresponding fused-operator implementation.

① **Rule schema:** Each rule specifies an ordered sequence of Velox operator types, together with predicates that the corresponding operators must satisfy. Listing 1 shows a rule for a final `Aggregation` immediately followed by an `OrderBy`. This rule requires a contiguous two-operator window consisting of a final aggregation with two grouping keys followed by an order-by. The empty predicate list on `OrderBy` means that the generic matcher does not additionally filter on sort-key metadata. The fused operator generated for this pipeline is nevertheless specialized to the output schema and ordering semantics of the original plan. Its output must match the

unfused Velox/cuDF reference: grouping keys and count-like outputs are compared exactly, null masks and schemas must match exactly, and ordered outputs are compared positionally. If the generated operator cannot be constructed for the candidate window, or if it fails equivalence, the original aggregation and order-by operators are executed through the cuDF fallback path.

Predicates use a composable `field:op:value` token format evaluated against a property bag extracted from each plan node. These predicates are structural, i.e., they inspect operator metadata such as the aggregation step, the number of grouping-key expressions, or the number of aggregate calls, but they do not inspect input tuples, column values, runtime cardinalities, or data distributions. For example, `groupingKeys.count:min:1` requires the aggregation operator to contain at least one grouping-key expression. It therefore matches grouped aggregations such as `GROUP BY c_custkey` or `GROUP BY nation, region`, but not a scalar aggregation such as `SELECT COUNT(*) FROM lineitem`. Tokens within a predicate list are AND-combined; nested lists denote OR-groups. Concretely, the supported op set is {`eq`, `neq`, `min`, `max`, `in`, `contains`} and `value` is parsed as a literal; field names are dotted paths into the property bag, so for example `aggregates[0].functionClass:in:[Sum,Count]` matches an aggregation whose first aggregate is a sum or count.

② **Registry and self-registration:** A singleton `FusionRuleRegistry` maps rule names (`"Q1Pipeline0"`, `"FilterProjectGroupByAgg"`, ...) to `FusionRuleFn` implementations that, given a contiguous window of Velox operators, return a `FusionMatch` describing the replacement fused operator. Each fused operator class self-registers at static-initialization time via a single macro placed in its `.cpp`. The registry therefore enumerates *every rule compiled into the binary*, while the JSON config selects *which subset is active for a given run.*

③ **Kernel replacement and execution:** At driver-compilation time, `FusionManager::apply()` walks each pipeline and, for each position, tries every enabled rule; on a hit it replaces the matched operator window with the fused operator before the normal per-operator cuDF lowering runs. Activation is therefore a runtime flag (`-fusion_rules=path/to/rules.json`), and omitting the flag disables fusion entirely. Kernel execution is achieved through a runtime bridge (§3.2).

A complementary `NodePropertyRegistry` decouples predicate evaluation from operator-specific code. Each Velox plan-node type registers an extractor that returns a flat property bag (`map<string, vector<string>>`); the matcher then evaluates `field:op:value` tokens generically against that bag. Adding predicate support for a new operator domain requires only registering one extractor — no change to the matcher, and new tokens become usable in JSON immediately.

Automatic rule generation: Authoring fusion dispatch rules by hand does not scale beyond a handful of patterns. We therefore derive fusion dispatch rules mechanically from the same query plans that Velox will later execute. For each TPC-H query, the benchmark driver first emits a serialized plan specification containing the operator DAG and the pipeline boundaries induced by blocking operators such as joins, exchanges, and final aggregations. The rule generator then traverses each pipeline in the plan, records the ordered

```jsonc
{
  "fusion_rules": [
    {
      // Registry key used by REGISTER_FUSED_OPERATOR(...,
      ↪ "Q1Pipeline1").
      "name": "Q1Pipeline1",

      // Matching-time switch.
      "enabled": true,

      // Pipeline boundary contract: no host<->device transfer is
      ↪ inserted
      // at either edge of the fused operator.
      "acceptsGpuInput": true,
      "producesGpuOutput": true,

      // Ordered Velox operator window. Operators must appear
      ↪ contiguously
      // and in this order.
      "operators": [
        {
          "type": "Aggregation",

          // Structural predicates on the aggregation slot.
          "predicates": [
             "step:eq:final",            // final aggregation stage
              "groupingKeys.count:eq:2",  // two-key GROUP BY;
              ↪ excludes global aggregation
             "aggregates.count:min:1"     // at least one aggregate
              ↪ call, e.g., COUNT(*)
          ]
        },
        {
          "type": "OrderBy",

          // No additional matcher-level predicates are used for
          ↪ OrderBy. The generated fused operator remains bound
          ↪ to the sort-key, direction, null-ordering, and
          ↪ output-schema assumptions of this pipeline;
          ↪ unsupported bindings fall back.
          "predicates": []
        }
      ]
    }
  ]
}
```

Listing 1: Auto-generated JSONC dispatch rule for a final-aggregation → order-by pipeline. Empty predicate lists indicate that no additional matcher-level predicates are used for that operator; operator-specific semantic assumptions are enforced by the generated fused-operator scaffold and equivalence gate.

sequence of Velox operator types, and queries the corresponding node-property extractors to obtain the structural attributes needed for matching.

The generated rules used in our experiments are structural and data-independent. They are best understood as dispatch rules that identify candidate Velox/cuDF operator windows, not as complete semantic specifications of the fused kernels. Predicates are evaluated only over property bags extracted from Velox plan nodes, such as operator type, aggregation stage, grouping-key count, aggregate-call count, and aggregate-function class when exposed by the corresponding property extractor. They do not inspect input tuples, value distributions, key cardinalities, selectivities, or runtime profiles.

Each fusion dispatch rule is associated with a broader soundness contract that specifies the class of Velox/cuDF operator windows for which the corresponding fused implementation is valid. This contract includes the matched operator sequence, supported

physical input types, nullability assumptions, grouping-key and aggregate-function assumptions, ordering requirements, output schema, comparison policy, and fallback behavior. Only the portion of this contract that is exposed by the current plan-node property extractors is encoded directly as JSON predicates. The remaining conditions are enforced by the generated scaffold, fused-operator construction checks, parameter binding, and the build and equivalence gates.

An empty predicate list on an operator therefore means that the matcher imposes no additional generic predicate for that operator beyond its type and position in the matched window. It does not mean that the fused implementation is semantically generic over all possible variants of that operator. For example, an `OrderBy` node with an empty predicate list is still tied to the sort-key, direction, null-ordering, and output-schema assumptions of the generated fused operator. If a candidate window cannot be bound to the generated operator interface, or if it fails the build or equivalence gates, the fusion is not deployed and the affected region executes through the Velox/cuDF fallback path.

A single generation pass produces 90 per-pipeline rule files covering all 22 TPC-H queries. We exclude Q1 from all comparisons because it is used as a worked prompt example; the per-query coverage and speedup numbers reported throughout §5 are therefore over the remaining 21 queries. For each pipeline, the generator also emits the corresponding fused-operator scaffold and registry entry, so that the generated rule name, the C++ operator stub, and the runtime registry remain synchronized.

3.2 Extending Velox with Triton Kernels

We extended Velox to host *Triton* kernels alongside cuDF ones, using Triton as a Python DSL in which the full body of such a pipeline can be written as one `@triton.jit` function. Triton is attractive here for three reasons: (i) it gives us register- and shared-memory-level control without writing CUDA C, (ii) its autotuning primitives let the generation pipeline explore block-size / warp-count / pipeline-stage configurations per query and per GPU without hand-tuning, and (iii) kernels live as self-contained single-file Python modules that evolutionary frameworks (§4) are designed to mutate. All Triton kernels reported in this paper — seeds, refined candidates, and the expert reference of §6.3 are deployed under `@triton.autotune` over a uniform configuration space, with the autotuner cache key fixed across runs.

Kernel-agnostic runtime bridge. The extension is intentionally kernel-agnostic: nothing in the core runtime mentions TPC-H. A C++ `TritonKernel` class wraps the CUDA Driver API which owns the loaded module, and exposes a named-parameter interface. It is given an ordered parameter layout — a list of `(name, c_type)` pairs with `c_type` in {`ptr`, `i32`, `i64`, `f32`, `f64`, `f16`} — from which it computes naturally-aligned byte offsets and allocates a single parameter buffer. Callers then set parameters by name (`kernel.set("A_ptr", d_A)`) and launch with a grid; a type-size check at `set()` time rejects mismatched widths before they can corrupt the launch buffer.

A launcher owns one `TritonKernel` and, on first use, delegates kernel compilation to a Python bridge via an embedded interpreter. The bridge, `jit_bridge.py`, is itself kernel-agnostic: given a dotted module name, a target compute capability, and a set of constexpr values, it imports the module, compiles `module.kernel` with the module-supplied `SIGNATURE`, and returns a dict of {cubin, kernel_name, shared_mem, num_warps, param_layout}. The C++ side stores the cubin and hydrates a `TritonKernel`; subsequent launches bypass Python entirely. To eliminate per-launch string lookups on the hot path, the launcher pre-computes byte offsets for every parameter once after compilation and writes directly into the raw parameter buffer thereafter.

4 AUTOMATED OPERATOR FUSION GENERATION AND EVOLUTIONARY REFINEMENT

Even for a skilled engineer, authoring a functionally correct and performant fused Triton kernel for all pipelines by hand would take a substantial amount of time and automating this task end-to-end with existing tools remains extremely difficult; instead, we stage the process as three pipeline-style tools, each gated by a machine-checkable bar so that the next stage starts from a known-good artifact.

4.1 Template- and Rule-driven Scaffolding

As described in §3.1, for every pipeline, both a structural fusion dispatch rule and a C++ operator scaffold with its `REGISTER_FUSED_OPERATOR` macro are pre-populated from Jinja2 templates. At the end of Stage 1 every pipeline therefore has a registered, matchable operator whose kernel body is still a `TODO`.

4.2 Operator Fusion Generation

We generate seeds using an agentic harness which interacts with a LLM in an iterative loop against a remote build/test service with two machine-checkable gates:

(1) **Build gate.** The patch is statically preflighted, applied on top of a pinned base SHA, pushed to a sandbox reference, and built. We detail the initial prompt and feedback mechanism in the following.
(2) **Equivalence gate.** On build success, the same instance runs a fused equivalence test, which compares the fused operator's output against the unfused discrete-operator pipeline on TPC-H data (SF=10)[1]. This also includes a correctness stress suite constructed to exercise SQL corner cases that are underrepresented in the benchmark data: null-heavy inputs, extreme key skew, decimal and overflow-sensitive values, empty inputs, tiny inputs, and large grouped aggregations. Each stress case is executed through both the fused path and the unfused Velox/cuDF reference path, and the fused candidate is accepted only if all cases agree under the same output-comparison policy.

The first attempt for each pipeline assembles a prompt with three concentric layers of context. The outermost layer is a fixed system prompt describing the contract the LLM must satisfy: the

[1]For integral, boolean, string, date, and key columns, comparison is exact. For floating-point aggregate outputs, comparison uses absolute tolerance 1e-4. Null masks and output schemas must match exactly. Ordered outputs are compared positionally; unordered grouped outputs are canonicalized before comparison. The unfused cuDF reference also passes against a DuckDB reference run on a held-out sample of the same queries, providing an independent semantic anchor for the comparison oracle.

Table 1: LLMs considered for automatic code generation. Costs are approximate API prices in USD.

Model	Cost ($) / MTok (Inp/Out)	Max Context Length	Open Source
Claude-Opus-4.7	$5 / $25	1M	✗
GPT-5.3-Codex	$1.75 / $14	400K	✗
Claude-Haiku-4.5	$1 / $5	200K	✗
Qwen3-30B-A3B-Thinking[†]	$0.20 / $2.40	256K	✓

[†] Self-hosted using vLLM 0.11.0. The checkpoint was downloaded from Hugging Face and deployed on October 29, 2025 with `SERVED_MODEL_NAME=Qwen/Qwen3-30B-A3B-Thinking-2507`. The listed cost range reflects public hosted API prices on OpenRouter [34], rather than our self-hosted cost.

writable-path allow-list (kernel `.py`, launcher `.cpp/.h`, and the per-pipeline fused-operator class), banned imports and shell-execution primitives, and the file-blocks output format that lets us compute the diff against a pinned base SHA rather than asking the LLM to produce valid unified-diff syntax.

The middle layer is the per-pipeline state at the base SHA: the rule JSON whose operator sequence the fused class must replace, the existing skeleton of the fused operator class with its `TODO` body, and the `CMakeLists.txt` fragments the LLM may extend. The innermost layer is a curated set of reference implementations for every cudf-side velox operator the rule names. For a `HashProbe + HashBuild` chain, that means the full source of `CudfHashJoinProbe`, `CudfHashJoinBuild`, and the `CudfHashJoinBridge` they communicate through, extracted via tree-sitter from the BASE_SHA velox checkout so the LLM sees the actual API surface it must mimic. Canonical Q1 fused pipelines are also provided as worked examples.

After the first attempt every subsequent prompt is conditioned on the previous attempt's failure. The harness classifies failures into a small taxonomy: `preflight`, `apply`, `push`, `compile`, `equivalence`, `equivalence_hang`, `benchmark`, and selects a heading and framing per kind so a corrupt-patch failure is not framed as a compile error and vice versa, which we found materially improved the LLM's recovery rate when the same diff syntax bug had repeated across attempts. For build and equivalence failures the harness fetches the full task log and runs a regex-driven extractor that anchors on gtest `[ FAILED ]` blocks, C++ exception headers, and `Value of:/Actual:/Expected:` assertion lines, keeping a small trailing context window (8 lines) around each anchor; remaining lines containing the substring `error` are deduplicated and capped at thirty entries.

The same extractor preserves `LOG(INFO)` lines emitted from the LLM's own `CudfQNPipelineMFused` class and from velox's cudf-exec operators, routed into a separate "diagnostic logs" section in the next iteration's prompt. This last channel closes the loop on cross-system invariants that the LLM cannot otherwise observe. For failures caused by mismatches between objects constructed by the generated code and the values later consumed by another subsystem, the LLM can add targeted `LOG(INFO)` statements at both sides of the interface. The extractor preserves these logs and routes them into the next prompt's diagnostic block, allowing the LLM to compare the generated-side state with the runtime-observed state rather than relying only on a downstream exception or assertion failure. We additionally surface the inter-attempt diff between the previous two patches so the LLM can see what it changed last round and avoid oscillating between two near-identical wrong answers.

The loop terminates when both gates pass (and a seed is generated) or when an iteration budget is exhausted. Passing fused equivalence at this stage is a stricter bar than a literal scaffold-compile and lets the next stage start directly in a mutation scope that assumes interface-level correctness.

4.3 Evolutionary Refinement with `SkyDiscover`

Seeds are correct but typically not optimally performant: they often default to a single-program grid-stride loop, pick safe but non-optimal block sizes, and miss GPU-specific micro-optimizations (eviction policies, one-hot grouping tricks, shared-memory use). Stage 3 hands each seed to `skydiscover` [27], a thin driver that routes the harness's evaluator/preflight/build–equivalence–benchmark stack to an external evolutionary search backend (OpenEvolve [48] or ShinkaEvolve [24]). The driver materializes the seed into a scratch checkout, wraps the target kernel file (`qN_fused.py`) in `EVOLVE-BLOCK-START/END` markers, and hands the backend a single evolvable file plus an evaluator. Evolutionary refinement is restricted to the Triton kernel body. The Velox operator interface, rule predicates, parameter layout, and equivalence/benchmark harness are held fixed, and every candidate must pass the same build and equivalence gates before it can contribute a benchmark result. The net effect is a staged pipeline; templates produce scaffolds, LLM loops produce correct seeds, evolutionary search produces fast kernels in which each stage's artifact is the input to the next, and each transition is gated on an automatic, machine-checkable criterion.

5 EXPERIMENTAL SETUP

5.1 Models

Table 1 summarizes the LLMs considered in our automatic code-generation pipeline. The set includes frontier proprietary models, a lower-cost proprietary model, and an open-source coding model, allowing us to compare both model capability and cost-performance trade-offs. During experimentation, when they are used, we log the number of input and output tokens and compute costs using the values reported in the aforementioned table.

5.2 Hardware

All measurements reported in this paper are collected on a single NVIDIA H100-SXM GPU with 80 GB of HBM3 memory, 3 TB/s peak memory bandwidth, and 132 SMs. The GPU is installed in a dual-socket Intel Xeon Platinum 8480+ host with 512 GB DDR5 memory. The system runs Ubuntu 22.04, NVIDIA driver 590.48.01.

5.3 Software Versions

All experiments use the same software stack across seed kernels and evolved candidates. The evaluated stack consists of Velox based on commit `bb9995d` with only changes to the fusion system and

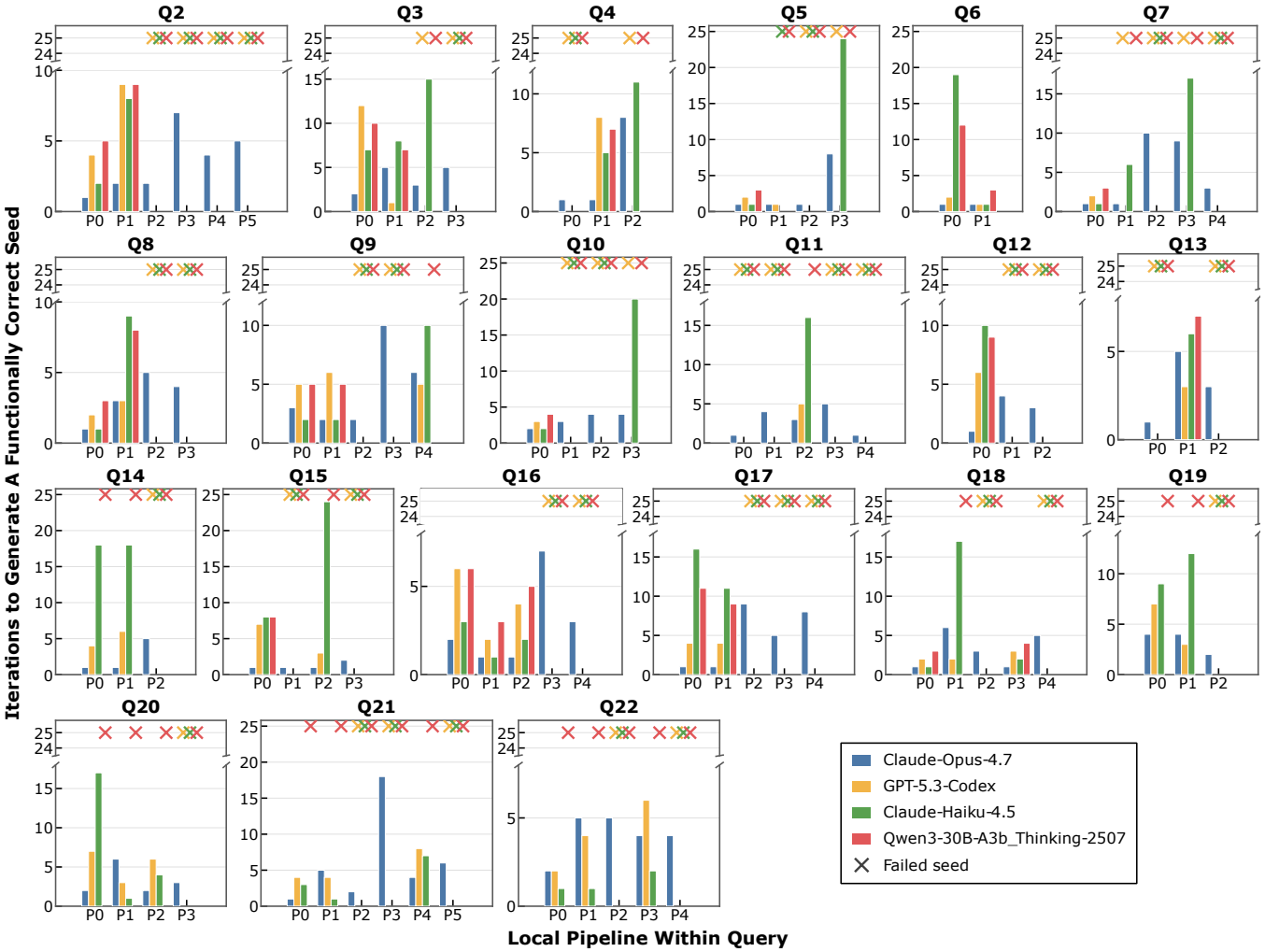

Figure 3: Per-query pipeline comparison across language models. Each panel corresponds to one TPC-H query, and each x-axis group denotes a distinct local pipeline for that query. Colored bars show the number of iterations required to obtain a successful seed for each model; failed runs that exhausted the iteration budget (25) are marked with an × at the budget limit.

kernel agnostic runtime bridge, cuDF 26.04, CUDA 13.1, Triton 3.6.0, SkyDiscover on commit `c0f6b70`, and Python 3.12.

5.4 Benchmarking

When benchmarking runtime, for every operator, including seeds, we execute one warm-up run followed by five timed runs on an otherwise-idle GPU. For all experiments, we observe the standard deviation across the five timed runs is ≤5%. Before benchmarking, every candidate must pass `fused_equivalence` against the unfused per-operator reference; candidates that fail equivalence are discarded without a timing result. The equivalence gate includes both the TPC-H SF10 reference run and the stress-test suite described in §4.2. Speedups are always reported for the end-to-end query runtime, relative to the per-operator cuDF lowering on the same hardware, using the same warm-up and timing protocol. All speedups in this section are computed only from fused configurations that passed equivalence against the unfused Velox/cuDF reference at the reported scale factor.

Failed seed generation does not make the query unexecutable. If a pipeline does not obtain a valid fused seed, or if no enabled fusion dispatch rule matches a region of the plan, that region executes through the existing per-operator Velox/cuDF backend. Thus, the opportunistic-fusion configuration enables all successfully generated and equivalence-passing fused pipelines, while all remaining pipelines fall back to the baseline cuDF path.

6 RESULTS

All speedups are normalized to the per-operator Velox/cuDF baseline. We avoid raw cross-system speedups and direct comparisons to prior work, instead isolating automated fusion within a fixed execution stack. Because the cuDF path shares the planner, vector

layout, and CPU–GPU boundary with the fused path, it provides a calibrated lower bound and attributes measured speedups to fusion rather than engine-level differences. We include a small expert-vs-generated comparison in §6.4. Although the evaluated rules are generated from TPC-H plans, the matcher uses only structural plan properties and not data values, selectivities, runtime profiles, or scale-factor-specific information; the same rule can therefore match any plan window satisfying the same operator and interface constraints.

6.1 Generating Correct Fused Operators

It is critical to automatically produce kernels that are not merely syntactically valid, but also build inside Velox and interface the unfused operator pipeline to produce functionally equivalent output. For a maximum of 25 iterations, Fig. 3 reports the number of iterations required to obtain the first seed that passes both gates for each TPC-H pipeline, and Table 2 summarizes success rates, iteration counts, and generation cost.

Claude-Opus-4.7 is the most reliable generator in this setting. It produces a correct seed for every evaluated pipeline, requiring 3.5 iterations per pipeline on average. The other models exhibit a substantially longer repair tail. GPT-5.3-Codex and Claude-Haiku 4.5 succeed on 47.7% and 53.4% of pipelines, respectively, while Qwen3-30B-A3B-Thinking-2507 succeeds on only 27.3%. The difference is visible in Fig. 3: failed runs cluster near the iteration budget, indicating that many unsuccessful attempts do not fail because of a single missing fix, but because the model cannot reliably satisfy the combined Triton, Velox, and cuDF interface constraints.

The model-inference cost results in Table 2[2] show that lower token price does not directly translate into lower cost per usable seed. As an example, Claude-Opus-4.7 has the highest average cost per iteration, but its short repair trajectories keep its cost at $1.17 per pipeline. This is close to GPT-5.3-Codex ($1.04) and Claude-Haiku-4.5 ($0.86), despite their lower per-token prices. Qwen3-30B-A3B-Thinking-2507 is the cheapest at $0.22 per pipeline, but its low success rate leaves most fusion opportunities without a valid seed[3]. We observe the dominant cost factor is therefore not only model price, but the ability to converge under build and equivalence feedback.

These results suggest that automatic fused operator generation is primarily an integration problem rather than a pure code-completion problem. Successful models must implement the Triton kernel, construct the correct Velox-side operator interface, respect cuDF-compatible layout assumptions, and use compiler and equivalence failures to repair cross-layer mistakes.

6.2 Performance of Generated Seeds

Fig. 4 reports end-to-end query speedups for all valid seed runs, grouped by model and scale factor. We refer to a single contiguous operator window matched by one fusion dispatch rule within a query plan as a *local pipeline*: there can be several per query, each independently fused. The top row enables one synthesized local pipeline at a time; the bottom row enables all successfully generated fused pipelines for the query. Interestingly, on average, we observe significant speedups from correct seeds, which were generated with the primary objective to be *functionally correct*, not performant. When only one fused pipeline is enabled per query, Claude-Opus-4.7 reaches mean speedups of 1.91×, 1.67×, and 2.96× at SF100, SF300, and SF1000, respectively. Claude-Haiku-4.5 reaches 2.18×, 2.41×, and 2.25×. However, not every valid seed improves performance in isolation: GPT-5.3-Codex and Qwen3-30B-A3B-Thinking-2507 include sub-baseline cases at SF100, with mean speedups of 0.89× and 0.74×.

These cases are consistent with the expected trade-off in fusion: removing launches and intermediate materialization can help, but conservative tiling, excessive live state, or poor launch parameters can offset the benefit for an individual pipeline[4]. The opportunistic-fusion configuration is more consistently beneficial. Enabling all available fused pipelines improves, on average, over the corresponding single-pipeline configuration for every model and scale factor in Fig. 4. Note that for some opportunistic-fusion configurations, a reduction in speedup is observed, compared to the equivalent pipeline case. We attribute this due to memory pressure and other effects, e.g., due to pipelined execution, which warrant further dissection in future work.

Claude-Opus-4.7 reaches 2.32×, 2.65×, and 3.19× mean speedup at SF100, SF300, and SF1000. Claude-Haiku-4.5 reaches 2.26×, 2.56×, and 2.66×, while GPT-5.3-Codex and Qwen3-30B-A3B-Thinking-2507 improve to 1.46–1.97× and 1.31–1.86×, respectively. Thus, even when an individual fused pipeline has limited impact, composing several local fusions within the same query can produce substantial end-to-end gains. The speedups also persist as the input scale increases. In the opportunistic-fusion configuration, each model achieves higher mean speedup at SF1000 than at SF100. This trend is consistent with the mechanism targeted by fusion: larger inputs amplify the cost of writing and rereading intermediate columns between GPU operators, so avoiding those materializations becomes more valuable.

6.3 Effect of Evolutionary Refinement

The seed-generation stage produces functionally correct fused operators, but these implementations are not necessarily locally optimal. We therefore evaluate OpenEvolve and ShinkaEvolve through SkyDiscover as a post-generation refinement stage. Both backends maximize a single scalar fitness $S \in [0, 1]$ that the harness assigns to each candidate after running its preflight, build, equivalence, and benchmark stages:

$$S = w_p\, p + w_b\, b + w_e\, e + w_s\, e \cdot \mathrm{clamp}\left(\frac{\sigma}{\sigma_{\max}}, 0, 1\right), \qquad (1)$$

where $p, b, e \in \{0, 1\}$ indicate that the candidate's patch applied, built, and passed equivalence against the unfused per-operator reference, σ is the measured speedup over the cuDF baseline (median of hot iterations), and $\sigma_{\max} = 5$ is a saturation cap that prevents a single outlier candidate from dominating selection pressure and starving exploration of other improvement directions. We use

[2]The costs in Table 2 exclude build time, equivalence-test time, benchmark time, GPU time, and orchestration overhead.

[3]The average cost per pipeline includes costs for failed seeds.

[4]Spot-checks with NVIDIA Nsight Compute on representative sub-baseline seeds attribute the regression primarily to (i) launch parameters that under-occupy the H100 SMs (e.g., a single-program grid-stride loop with insufficient blocks to fill the device) and (ii) increased per-thread live state that drops achieved occupancy below the unfused baseline.

Table 2: Aggregate model-level performance and cost for automated operator fusion. For each language model, we report build-gate success, end-to-end seed success, average iterations per pipeline, and average USD cost per iteration and per pipeline across the generated TPC-H fused-operator pipelines. The denominator includes both successful and failed seed-generation attempts; failed pipelines are not dropped from the coverage statistics and execute through the Velox/cuDF fallback path when evaluating full-query configurations. Fig. 3 shows the corresponding per-query and per-pipeline coverage.

Model	Successful Build Rate (%)	Seed Success Rate (%)	Avg Iterations/ Pipeline	Avg Cost (USD)/ Iteration	Avg Cost (USD)/ Pipeline
Claude-Opus-4.7	57.4	100.0	3.5	$0.33	$1.17
GPT-5.3-Codex	26.3	47.7	15.12	$0.07	$1.04
Claude-Haiku-4.5	27.2	53.4	15.99	$0.05	$0.86
Qwen3-30B-A3B-Thinking-2507	14.2	27.3	19.9	$0.01	$0.22

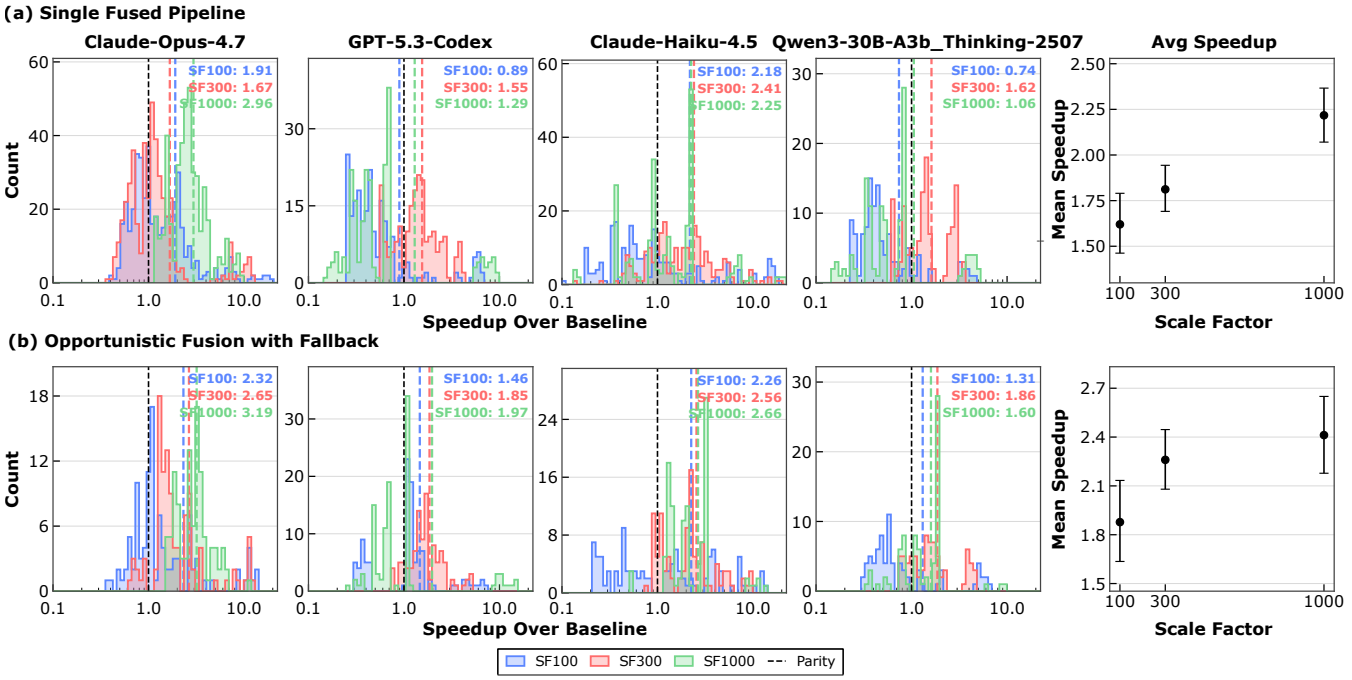

Figure 4: Distribution of end-to-end query speedups over the baseline execution, stratified by model and scale factor. (a) The top row shows speedups for individually synthesized local pipelines, while (b) the bottom row shows speedups when all successfully generated fused pipelines are adopted and failed or unmatched regions fall back to cuDF. In the rightmost column, we summarize the mean speedup across all models. Error bars denote bootstrap 95% confidence intervals.

$(w_p, w_b, w_e, w_s) = (0.05, 0.15, 0.30, 0.50)$; together with the short-circuits below, these enforce a strict failure ordering (forbidden < patch-drift < build-failed < equivalence-failed < equivalent). Pre-flight rejections (forbidden path or scope violations) and patches that do not apply are short-circuited to $S = 0$ and $S = 0.01$, respectively. Within the ordering constraint $w_p + w_b + w_e = 0.5$, which guarantees any equivalence-passing candidate strictly outranks every broken candidate regardless of measured speedup, the exact weights were chosen empirically and we did not observe sensitivity to small perturbations.

Fig. 5 reports estimated best-so-far trajectories for four selected SF1000 seeds, chosen to cover different seed-availability regimes: Q10P0, which was successfully generated by all four models; Q11P2, which was successfully generated by Claude-Opus-4.7, Claude-Haiku-4.5, and GPT-5.3-Codex; Q10P3, which was successfully generated by Claude-Opus-4.7 and Claude-Haiku-4.5; and Q10P1, which was successfully generated only by Claude-Opus-4.7.

Each curve starts from the measured seed speedup at iteration 0. Improvements are shown as step functions because most valid refinement attempts preserve the incumbent performance, while occasional mutations yield a new best candidate. After 25 refinement iterations, the largest projected endpoint is observed for Q10P3: 6.02× with OpenEvolve and 6.09× with ShinkaEvolve over the per-operator Velox/cuDF baseline. Across the selected pipelines, the

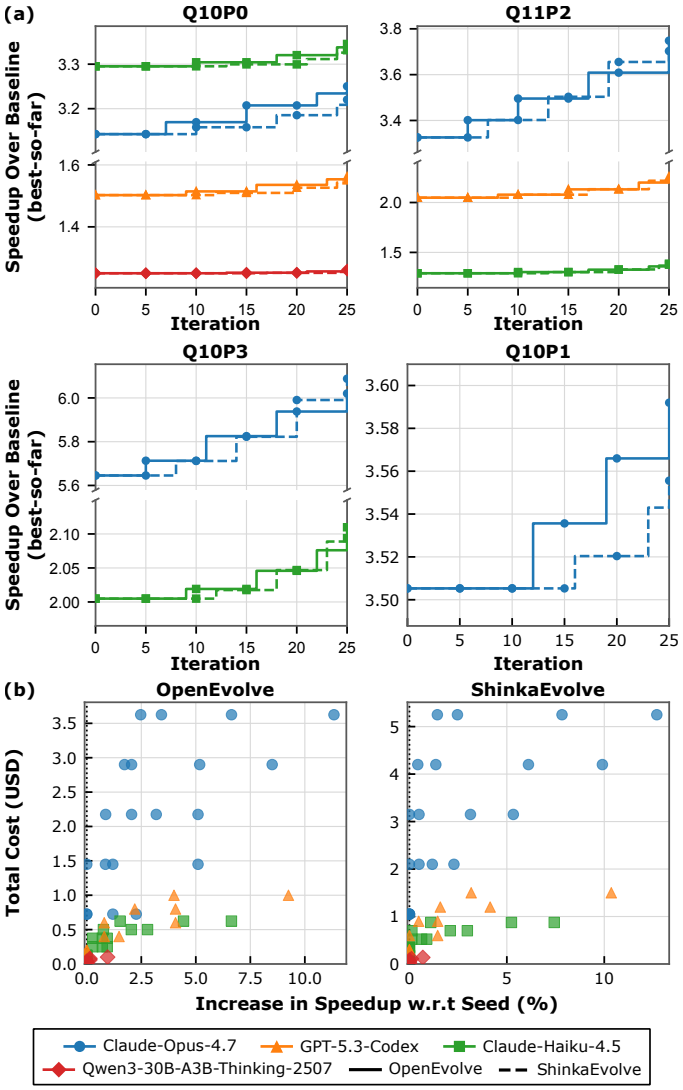

Figure 5: Evolutionary kernel refinement trajectories for selected TPC-H seeds. Speedups are reported for SF1000. (a) The first four panels report best-so-far speedup over the per-operator Velox/cuDF baseline as a function of search iteration, starting from the model-generated seed speedup at iteration 0. (b). The bottom-row cost panels show total estimated generation-plus-refinement cost versus percentage increase in speedup relative to the iteration 0 seed.

final best-so-far speedups are 3.35×/3.33× for Q10P0, 3.59×/3.56× for Q10P1, 6.02×/6.09× for Q10P3, and 3.70×/3.75× for Q11P2 with OpenEvolve/ShinkaEvolve, respectively. Three of the four trajectories are essentially flat after iteration 0, which means kernel-level wins are unevenly distributed across pipelines and depend on the algorithmic headroom left by the seed.

The bottom row of Fig. 5 reports refinement-only cost as a function of improvement over the iteration 0 seed. The results illustrate the cost–quality tradeoff of post-generation search: stronger models produce larger expected improvements but at higher refinement cost, while lower-cost models provide cheaper but more limited exploration.

Table 3: Composition of the selected refinement targets. The set includes both multi-operator fusion windows and single-operator specialization windows.

Target	Operator window	Multi-Operator
Q10P0	`CudfGroupby` → `CudfOrderBy` → `CudfFilterProject` → `CudfLimit`	✓
Q10P1	`CudfFilterProject` → `CudfHashJoinProbe` → `CudfGroupby`	✓
Q10P3	`CudfHashJoinBuild`	✗
Q11P2	`CudfHashJoinProbe`	✗

6.4 A Case Study: Expert-vs-Generated Calibration

The end-to-end speedups in §6.2 and §6.3 are reported against the per-operator Velox/cuDF baseline. To calibrate these results against human-written code, we add expert hand-tuned Triton references for two representative targets. The first, Q10P3, is a degenerate single-operator window and therefore measures plan-specific specialization. The second, Q10P0, is a genuine multi-operator fusion window and therefore directly evaluates the central claim of this work: that generated fused kernels can approach expert implementations for bounded SQL operator sequences.

Table 4 reports both comparisons. Q10P3 matches a single `HashBuild` operator, so its speedup should be interpreted as plan-specific HashBuild specialization rather than operator fusion. Q10P0, in contrast, spans a `CudfGroupby` → `CudfOrderBy` → `CudfFilterProject` → `CudfLimit` window and therefore provides a direct expert-vs-generated calibration point for multi-operator fusion.

The expert references are hand-written Triton implementations authored independently of the LLM- and evolutionary-search outputs. Each expert variant uses the same input/output contract, output schema, ordering semantics, null-mask requirements, equivalence gate, benchmark protocol, and `@triton.autotune` configuration space as the corresponding generated and refined variants.

For Q10P3, the Claude-Opus-4.7 seed reaches 89.0% of expert performance, suggesting that the generated kernel captures most of the available single-operator specialization benefit. Evolutionary refinement closes much of the remaining gap, improving the seed by 6.5% with OpenEvolve and 7.8% with ShinkaEvolve, and bringing both refined candidates within roughly 4–5% of the expert implementation. The generated seed for Q10P0 reaches 84.0% of expert performance, while OpenEvolve and ShinkaEvolve improve this to 89.6% and 89.0%, respectively. The larger seed-to-expert gap for Q10P0 is expected: when a target spans more than one operator, the optimization space is broader than for a single-operator specialization. In particular, an expert implementation can co-optimize

Table 4: Expert-vs-generated calibration at SF=1000. Q10P3 is a single-operator HashBuild specialization, while Q10P0 is a genuine multi-operator fusion window: `CudfGroupby` → `CudfOrderBy` → `CudfFilterProject` → `CudfLimit`. All Triton variants are reported under `@triton.autotune` with the same configuration space; the per-operator Velox/cuDF baseline serves as the normalization point for each target.

Target	Variant	Speedup over cuDF	% of Expert
Q10P3	Per-operator cuDF baseline	1.00×	15.7%
	Claude-Opus-4.7 seed	5.65×	89.0%
	OpenEvolve refined	6.02×	94.8%
	ShinkaEvolve refined	6.09×	95.9%
	Expert hand-tuned	6.35×	100%
Q10P0	Per-operator cuDF baseline	1.00×	26.7%
	Claude-Opus-4.7 seed	3.14×	84.0%
	OpenEvolve refined	3.35×	89.6%
	ShinkaEvolve refined	3.33×	89.0%
	Expert hand-tuned	3.74×	100%

intermediate-value flow, avoid materializing columns that are consumed only within the fused window, and combine ordering, projection, and limit handling in a way that is difficult for the initial seed to discover. Thus, Q10P0 shows both that generated multi-operator fused kernels can approach expert performance and that such windows leave meaningful optimization headroom beyond the initial generated seed.

7 RELATED WORK

Automatic kernel generation and fusion. There is a large existing literature on automatic kernel generation and operator fusion for deep-learning inference and tensor programs. Tensor compilers such as TVM [7], XLA [45], TASO [15], Ansor [58], and Halide [43] expose graph-, schedule-, or tile-level representations for fusion, autotuning, and accelerator code generation. Recent LLM- and search-driven systems push kernel automation further, using executable feedback to generate or optimize kernels, algorithms, and parallel code [2, 23, 32, 33, 35, 42]. Our workflow borrows this generate–test–refine structure, but changes both the artifact and the oracle: the generated code implements relational operator windows inside Velox and is checked against an unfused SQL execution path rather than against tensor kernels alone. These systems do not address the integration problem that dominates our setting: a fused candidate must (i) bind to a host-side C++ operator interface, (ii) respect the cuDF column layout and ownership conventions, and (iii) produce schema- and ordering-correct output against a streaming Velox pipeline.

GPU-accelerated data processing. GPU database systems have long demonstrated the benefits and limitations of accelerator execution for analytics. Early work mapped relational operators to GPUs [12]; Ocelot, Ocelot/HyPE, and CoGaDB explored hardware-oblivious or adaptive CPU/GPU execution [4, 5, 13]; and surveys identify data movement, memory management, and operator-at-a-time execution as recurring issues [6]. Full-system and product-oriented engines such as MapD/HeavyDB and RateupDB show the practicality of hybrid CPU/GPU analytics [25, 30]. More recent systems address concurrency, pipelining, and modern GPU behavior: MultiQx-GPU targets concurrent analytical queries, GPL and Pyper pipeline query execution on GPUs, Themis addresses warp-level imbalance, and recent studies analyze scaling limits on modern GPU platforms [14, 18, 26, 38, 39, 55]. Velox (the substrate of our system) and Sirius are closest in deployment style because they preserve a general query-engine abstraction while lowering supported work to accelerated backends [37, 40].

Maximus [17], like Velox/cuDF, lowers operators per-kernel behind a shared planner. cuDF itself recently added JIT-compiled transforms that fuse element-wise expressions within a single column transform [1]; this fuses at expression granularity inside one operator, whereas we fuse across whole `CudfOperator` boundaries (e.g., aggregation → order-by → limit). The two are therefore complementary rather than alternatives, and a direct comparison would conflate fusion scope with implementation.

Data Path Fusion (DPF) fuses I/O, decompression, and query processing into a GPU-driven data path [36]. DPF and our work share the goal of reducing fragmentation in GPU analytics, but operate at different boundaries. DPF redesigns the storage-to-query data path, whereas we operate inside an existing Velox/cuDF backend and replace only bounded in-pipeline operator windows with generated Triton kernels.

Query compilation and data-system fusion. Compiled query engines established pipeline-level code generation as a way to remove interpretation overhead and avoid unnecessary materialization. HyPer's produce/consume model generates tight CPU pipelines, and later work characterizes the trade-offs between compiled and vectorized execution [20, 31]. Kernel Weaver is a direct GPU predecessor: it fuses relational-algebra kernels using producer–consumer dependences to reduce intermediate traffic [57]. rNdN [22] is a canonical GPU query compiler that fuses operators via a deterministic compiler pass over an array IR; we instead treat fusion as generation-and-validation, with an LLM emitting Triton kernels gated by equivalence against the unfused path. Relaxed operator fusion and related compiler work show that maximal fusion is not always best, since staging can improve compilation time, scheduling, and resource use [3, 28]. Other systems expose query computation through templates or IRs, including LegoBase, Tupleware, Voodoo, Proteus, LingoDB, Weld, Flare, HetExchange, and HAPE [8–11, 16, 19, 21, 37, 41]. Crystal demonstrates the performance potential of hand-engineered GPU query kernels [46]. These systems rely primarily on hand-written templates, libraries, or deterministic compiler passes.

Workload-specific synthesis. Bespoke OLAP is the closest work at the workload-specific end of the design space, synthesizing complete one-size-fits-one analytical engines with LLM-driven refinement [56]. Bespoke OLAP and our work address complementary regimes: Bespoke OLAP synthesizes the entire engine for a known workload and accepts the cost of regenerating that engine when the workload changes, while our approach produces fused operators *inside* an existing Velox/cuDF backend so that unmatched plan windows continue to execute through the production engine. We do not claim that pipeline-granularity fusion outperforms full workload-specific synthesis on workloads where the latter is feasible; we instead claim that pipeline-granularity fusion targets the regime

where regenerating an entire engine is impractical (multi-tenant deployments, frequently-changing plans, partial GPU coverage) but where opportunistic fused kernels can still be deployed safely behind the cuDF fallback path.

LLMs in database systems. Recent work applies LLMs to the optimizer (LLM-QO [51], GenJoin [49]) and to knob tuning (GPTuner [52]). These target plan choice or configuration; we apply LLM generation at the execution layer.

8 CONCLUSION

We presented an automated workflow for generating fused GPU operators for SQL execution inside an existing Velox/cuDF backend. The workflow targets pipeline-granularity fusion: contiguous, non-blocking operator windows are identified through structural dispatch rules, replaced with Triton-backed fused operators, and validated against the unfused Velox execution path. Through an agentic harness, LLMs first produce functionally correct seed kernels under build and equivalence gates, and evolutionary search then refines only the Triton kernel body under the same correctness checks. Overall, the results support pipeline-granularity fusion as a practical middle point between per-operator GPU lowering and workload-specific engine synthesis: generated fused operators can be introduced opportunistically, validated against the existing execution path, and deployed with automatic fallback for regions that are unsupported or fail generation.

8.1 Future Work

The most directly motivated extensions are to extend the dispatch-rule layer to non-TPC-H workloads to validate structural predicates generalize beyond the patterns surfaced by a single benchmark and to discect the opportunistic-fusion regression cases observed in Fig. 4 into kernel-level (register pressure, occupancy) and pipeline-level (memory pressure, scheduler interactions) contributions. Additionally, we believe exploring a single-task agentic formulation, that jointly proposes and improves fused kernels rather than separating kernel addition from refinement is a promising future direction. A further direction is to explore fusion opportunities that require visibility into the full logical query plan rather than a bounded physical operator window. While pipeline-granularity fusion is effective for the contiguous non-blocking windows studied here, some optimizations (such as reordering joins across pipeline boundaries or eliminating redundant projections across blocking stages) can only be identified with a broader view of the plan. Combining physical fusion with logical plan rewriting could therefore yield additional end-to-end speedups beyond what either technique achieves in isolation. Finally, data distribution can substantially affect whether a fused kernel is the right specialization target. The structural dispatch rules used here are data-independent by design, which enables them to match any plan window satisfying the operator and type constraints regardless of input cardinalities or value distributions. However, when data is highly skewed (for example, a grouped aggregation over a column with a few dominant keys), a fused kernel tuned for the uniform case may perform poorly, and a distribution-aware specialization or a fallback to the cuDF path may be more appropriate. Exploring robustness-vs-specialization as an explicit design axis, and potentially incorporating lightweight runtime statistics into the dispatch decision, is an interesting direction for future work.

REFERENCES

[1] Basit Ayantunde, David Wendt, and Gregory Kimball. 2025. *Efficient Transforms in cuDF Using JIT Compilation.* https://developer.nvidia.com/blog/efficient-transforms-in-cudf-using-jit-compilation/ NVIDIA Developer Blog.

[2] Carlo Baronio, Pietro Marsella, Ben Pan, Simon Guo, and Silas Alberti. 2025. Kevin: Multi-Turn RL for Generating CUDA Kernels. abs/2507.11948 (2025). arXiv:2507.11948 doi:10.48550/ARXIV.2507.11948

[3] Matthias Boehm et al. 2018. On optimizing operator fusion plans for large-scale machine learning in systemML. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1755–1768. doi:10.14778/3229863.3229865

[4] Sebastian Breß et al. 2014. Ocelot/HyPE: optimized data processing on heterogeneous hardware. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1609–1612. doi:10.14778/2733004.2733042

[5] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1891–1906. doi:10.1145/2882903.2882936

[6] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. GPU-Accelerated Database Systems: Survey and Open Challenges. 15 (2014). doi:10.1007/978-3-662-45761-0_1

[7] Tianqi Chen et al. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (2018), Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594.

[8] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. doi:10.14778/3303753.3303760

[9] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. 2019. Hardware-conscious Query Processing in GPU-accelerated Analytical Engines. In *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019).

[10] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings* (2015).

[11] Grégory M. Essertel et al. 2018. Flare: optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 799–815.

[12] Bingsheng He et al. 2009. Relational query coprocessing on graphics processors. 34, 4 (2009), 21:1–21:39. doi:10.1145/1620585.1620588

[13] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. 6, 9 (2013), 709–720. doi:10.14778/2536360.2536370

[14] Kijae Hong, Kyoungmin Kim, Young-Koo Lee, Yang-Sae Moon, Sourav S Bhowmick, and Wook-Shin Han. 2024. Themis: A GPU-Accelerated Relational Query Execution Engine. *Proc. VLDB Endow.* 18, 2 (Oct. 2024), 426–438. doi:10.14778/3705829.3705856

[15] Zhihao Jia et al. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

[16] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow.* 15, 11 (July 2022), 2389–2401. doi:10.14778/3551793.3551801

[17] Marko Kabić, Shriram Chandran, and Gustavo Alonso. 2025. Maximus: A Modular Accelerated Query Engine for Data Analytics on Heterogeneous Systems. *Proc. ACM Manag. Data* 3, SIGMOD, Article 187 (2025). doi:10.1145/3725324

[18] Marko Kabić, Bowen Wu, Jonas Dann, and Gustavo Alonso. 2025. Powerful GPUs or Fast Interconnects: Analyzing Relational Workloads on Modern GPUs. *Proc. VLDB Endow.* 18, 11 (July 2025), 4350–4363. doi:10.14778/3749646.3749698

[19] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 972–983. doi:10.14778/2994509.2994516

[20] Timo Kersten et al. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. doi:10.14778/3275366.3284966

[21] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proc. VLDB Endow.* 7, 10 (June 2014), 853–864. doi:10.14778/2732951.2732959

[22] Alexander Krolik, Clark Verbrugge, and Laurie J. Hendren. 2023. rNdN: Fast Query Compilation for NVIDIA GPUs. 20, 3 (2023), 41:1–41:25. doi:10.1145/3603503

[23] Robert Tjarko Lange et al. 2025. Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization. abs/2509.14279 (2025). arXiv:2509.14279 doi:10.48550/ARXIV.2509.14279

[24] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution. abs/2509.19349 (2025). arXiv:2509.19349 doi:10.48550/ARXIV.2509.19349

[25] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The art of balance: a RateupDB™ experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endow.* 14, 12 (July 2021), 2999–3013. doi:10.14778/3476311.3476378

[26] Yinan Li et al. 2025. Scaling GPU-Accelerated Databases Beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (July 2025), 4518–4531. doi:10.14778/3749646.3749710

[27] Shu Liu et al. 2026. SkyDiscover: A Flexible, Adaptive Framework for AI-Driven Scientific and Algorithmic Discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems* (2026-05) *(CAIS '26)*. ACM, 1223–1227. doi:10.1145/3786335.3813221

[28] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 13 pages. doi:10.14778/3151113.3151114

[29] Meta Platforms, Inc. 2026. *Axiom*. https://github.com/facebookincubator/axiom GitHub repository.

[30] Todd Mostak. 2013. An Overview of \{}{MapD\{}} (Massively Parallel Database). (2013). Technicaloverview

[31] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. 4, 9 (2011), 539–550. doi:10.14778/2002938.2002940

[32] Daniel Nichols, Joshua H. Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can Large Language Models Write Parallel Code?. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 281–294. doi:10.1145/3625549.3658689

[33] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. abs/2506.13131 (2025). arXiv:2506.13131 doi:10.48550/ARXIV.2506.13131

[34] OpenRouter. 2026. *Qwen3 30B A3B Thinking 2507 – Pricing*. https://openrouter.ai/qwen/qwen3-30b-a3b-thinking-2507

[35] Anne Ouyang et al. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels?. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025* (2025) *(Proceedings of Machine Learning Research)*, Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (Eds.). PMLR / OpenReview.net.

[36] Tsuyoshi Ozawa and Kazuo Goda. 2026. Data Path Fusion in GPU for Analytical Query Processing. arXiv:2605.10511 [cs.DB] https://arxiv.org/abs/2605.10511

[37] Shoumik Palkar et al. 2018. Evaluating end-to-end optimization for data analytics applications in weld. 11, 9 (May 2018), 1002–1015. doi:10.14778/3213880.3213890

[38] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 202–214. doi:10.14778/3425879.3425890

[39] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-based Pipelined Query Processing Engine. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 1935–1950. doi:10.1145/2882903.2915224

[40] Pedro Pedreira et al. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3372–3384. doi:10.14778/3554821.3554829

[41] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1707–1718. doi:10.14778/3007328.3007336

[42] Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. 2026. CORAL: Towards Autonomous Multi-Agent Evolution for Open-Ended Discovery. abs/2604.01658 (2026). arXiv:2604.01658 doi:10.48550/ARXIV.2604.01658

[43] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. doi:10.1145/2491956.2462176

[44] RAPIDS Development Team. 2026. *RAPIDS cuDF*. https://docs.rapids.ai/api/cudf/stable/

[45] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[46] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. doi:10.1145/3318464.3380595

[47] Akash Shankaran, George Gu, Weiting Chen, Binwei Yang, Chidamber Kulkarni, Mark Rambacher, Nesime Tatbul, and David E. Cohen. 2023. The Gluten Open-Source Software Project: Modernizing Java-based Query Engines for the Lakehouse Era. In *VLDB Workshops*.

[48] Asankhaya Sharma. 2025. *OpenEvolve: An Open-Source Evolutionary Coding Agent*. https://github.com/algorithmicsuperintelligence/openevolve GitHub repository.

[49] Pavel Sulimov, Claude Lehmann, and Kurt Stockinger. 2025. GenJoin: Conditional Generative Plan-to-Plan Query Optimizer that Learns from Subplan Hints. *Proc. ACM Manag. Data* 3, 4, Article 247 (Sept. 2025), 25 pages. doi:10.1145/3749165

[50] Yutian Sun et al. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2, Article 189 (June 2023), 25 pages. doi:10.1145/3589769

[51] Jie Tan et al. 2025. Can Large Language Models Be Query Optimizer for Relational Databases? *Proc. ACM Manag. Data* 3, 6, Article 306 (Dec. 2025), 28 pages. doi:10.1145/3769771

[52] Jiale Tang, Yi Wang, Jiaqi Liu, Wenhui Tao, Pengfei Xu, Xuanhe Zhou, Bin Cui, Nan Tang, Yongjun Han, and Wei Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1939–1952. doi:10.14778/3659437.3659449

[53] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[54] Velox Contributors. 2025. *Velox Experimental cuDF GPU Execution Backend*. https://github.com/facebookincubator/velox/tree/main/velox/experimental/cudf GitHub repository.

[55] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. 7, 11 (2014), 1011–1022. doi:10.14778/2732967.2732976

[56] Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. 2026. Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines. abs/2603.02001 (2026). arXiv:2603.02001 doi:10.48550/ARXIV.2603.02001

[57] Haicheng Wu, Gregory Frederick Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2012, Vancouver, BC, Canada, December 1-5, 2012* (2012). IEEE Computer Society, 107–118. doi:10.1109/MICRO.2012.19

[58] Lianmin Zheng et al. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020). USENIX Association, 863–879.