Into the CUDA Multiverse of Runtime Compilation: Exploring Kernel Fusion Strategies for GPU Database Systems

Maha Alwahibi
University of Bamberg
Germany
maha.alwahibi@uni-bamberg.de

Bachana Jangebashvili
University of Bamberg
Germany
bachana.jangebashvili@stud.

Maximilian E. Schüle
University of Bamberg
Germany
maximilian.schuele@uni-bamberg.de

ABSTRACT

This demonstration presents the CUDA Code-General, a query compilation framework for GPU database systems. The interface allows selecting any of the 22 TPC-H benchmark queries or entering arbitrary SQL, to choose between runtime or static compilation and to inspect the generated CUDA code for three kernel fusion strategies side by side: fused multiple sequential kernels, cooperative groups, and dynamic parallelism. Users can observe how the number of generated pipelines affects compilation cost, how cooperative groups fuse them into a single kernel with synchronisation barriers, and how dynamic parallelism delegates kernel launches to the device.

KEYWORDS

Query Compilation, GPU Database Systems

PVLDB Reference Format:
Maha Alwahibi, Bachana Jangebashvili, and Maximilian E. Schüle. Into the CUDA Multiverse of Runtime Compilation: Exploring Kernel Fusion Strategies for GPU Database Systems. PVLDB, 19(12): 4658 - 4661, 2026.
doi:10.14778/3827998.3828090

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/MaxEmanuel/CUDACodeGeneral.

1 INTRODUCTION

Interpretation of query plans—the traditional Volcano-based execution model [7]—suffers from one function call per operator and tuple. The push-based execution model [8], which inverts the data flow, together with code generation eliminates interpreted function calls at execution time [1]. When porting this approach to GPUs, each pipeline within a query plan maps to a CUDA kernel that is generated and compiled at query time. GPU database systems such as DogQC [6], HetExchange [3] and Pyper [10] have demonstrated that code generation enables efficient tuple-at-a-time execution on GPUs, leveraging the *single instruction, multiple threads* (SIMT) execution model for data-level parallelism.

However, just-in-time compilation introduces a compilation overhead that is noticeable at interactive query latencies. Each pipeline within a query plan requires a separate kernel and the compilation cost scales linearly with the number of pipelines. A TPC-H query [5] with seven pipelines, for instance, incurs seven independent compilation invocations using NVIDIA's nvcc compiler. NVIDIA's runtime compiler (NVRTC) [9] mitigates this by compiling CUDA C++ source to PTX directly within the running process, avoiding the overhead of spawning an external compiler. Furthermore, multiple kernels can be compiled in a single NVRTC invocation through *fused compilation*, reducing the compilation cost from linear to near-constant in the number of pipelines.

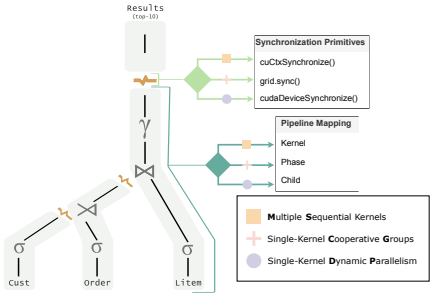

Figure 1: Kernel fusion strategies: Each pipeline maps to either a kernel (MS), a phase (CG) or a child kernel (DP).

Beyond compilation, the overhead of launching multiple kernels sequentially from the host can be reduced through kernel fusion. Cooperative groups, introduced with CUDA 9 on the Volta architecture, allow all pipelines to be fused into a single kernel with grid-level synchronisation barriers (`grid.sync()`) between phases. Dynamic parallelism, available since the Kepler architecture, enables a parent kernel on the device to launch child kernels, eliminating the host-side launch overhead. Both strategies allow all query phases to be compiled as a single compilation unit.

Our prior work [11] introduced runtime compilation for code-generating database systems. We present the CUDA Code-General as an extensible compilation framework that supports these strategies. We developed a web-based interface for the CUDA Code-General to demonstrate *how* generated CUDA code differs structurally between strategies, *what* the query plan looks like for a given SQL query and *where* the time is spent across compilation and execution. The interface allows users to select any of the 22 TPC-H queries or enter arbitrary SQL, inspect the generated CUDA kernels for three strategies side by side, examine the query plan as an operator tree and compare compilation and execution times.

This paper is structured as follows: Section 2 presents the architecture of the CUDA Code-General (cf. Figure 1), covering the kernel fusion strategies, runtime compilation and the SQL parser. Section 3 presents the web frontend and the demonstration scenarios. Sections 4 and 5 conclude the paper.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828090

2 SYSTEM OVERVIEW

The CUDA Code-General is an extensible query compilation framework for GPU database systems. It implements relational operators—table scans, selections, projections, joins (semi, anti, equi) and aggregations—according to the push-based, tuple-at-a-time execution model [8]. Each operator provides a `produce()` function that emits CUDA C++ code into a code-generation object, which assembles the final kernel source. A query plan with pipeline-breaking operators (such as joins) is decomposed into multiple pipelines, each of which maps to a generated kernel.

Fused Runtime Compilation with NVRTC. NVIDIA's runtime compiler NVRTC [9] compiles CUDA C++ source code to PTX assembly within the running process, without spawning an external compiler. The compilation process begins with `nvrtcCreateProgram()`, which accepts a C++ source string, followed by `nvrtcCompileProgram()` to produce PTX. The resulting PTX is loaded into a `CUmodule` via the CUDA Driver API and individual kernel functions are extracted by name using `cuModuleGetFunction()`.

In the traditional *multiple sequential kernels* (MS) strategy, each pipeline is compiled in a separate NVRTC invocation, incurring a fixed overhead of approximately 80–120 ms per invocation for parsing headers, initialising the compiler and generating PTX. A query with n pipelines thus requires $n \times T_{\text{compile}}$ of compilation latency. *Fused compilation* (MS_f) eliminates this redundancy by concatenating all kernel sources into a single compilation unit. The resulting `CUmodule` contains all kernels, from which each is extracted individually—a lookup costing microseconds rather than milliseconds. To ensure that `cuModuleGetFunction()` can locate each kernel by name without C++ name mangling, all kernels use `extern "C"` linkage. The launch code remains unaffected: it still receives individual `CUfunction` handles and launches them sequentially with independent grid sizes, block sizes and arguments.

Compared to invoking `nvcc` as a separate process, NVRTC achieves an order-of-magnitude reduction in compilation time. At scale factor 1, compiling TPC-H Query 8 (seven kernels) takes approximately 2.9 seconds with nvcc but only 420 ms with NVRTC using MS and 91 ms with MS_f—a 32× improvement.

Kernel Fusion with Cooperative Groups. Cooperative groups (CG), introduced with CUDA 9 on the NVIDIA Volta architecture, extend the CUDA programming model with thread synchronisation at grid, block and warp level. For query compilation, the key feature is grid-level synchronisation via `grid.sync()`, which acts as a barrier across all thread blocks of a kernel. This allows multiple query pipelines to be fused into a single kernel, with `grid.sync()` calls separating each phase. The generated code uses a `cooperative_groups::grid_group` object to synchronise all threads before proceeding to the next pipeline.

`cuLaunchCooperativeKernel()` from the Driver API launches a fused CG kernel, which requires all thread blocks to be co-resident on the GPU. This constrains the grid size to the hardware occupancy: on an NVIDIA RTX A2000 with 26 streaming multiprocessors (SMs) and a block size of 256, this yields at most $6 \times 26 = 156$ blocks. To compensate, each phase uses a grid-stride loop that allows the fixed number of threads to process arbitrarily large input tables. While this introduces a loop overhead compared to the data-dependent grid sizes of MS, it enables all intermediate data to remain resident in on-chip caches across pipeline boundaries.

Kernel Fusion with Dynamic Parallelism. Dynamic parallelism (DP), introduced with the NVIDIA Kepler architecture, allows a kernel running on the device to launch child kernels without returning to the host. In our framework, the host launches a single parent kernel, which then sequentially dispatches child kernels using the CUDA Runtime API, with `cudaDeviceSynchronize()` between phases to enforce data dependencies. Each child kernel has its own independently computed grid and block size, adapting the degree of parallelism to the input cardinality of each pipeline—matching the behaviour of MS.

Since DP compiles all child kernels together with the parent in a single compilation unit, it achieves compilation times comparable to MS_f and CG. Unlike CG, DP does not restrict the grid size, so child kernels can launch with the same data-dependent configurations as in MS. However, child kernels launched from the device incur a small device-side scheduling overhead and parent and child kernels do not share access to shared memory, as they execute in separate grids.

SQL Parser and Planner. In addition to the 22 hardcoded TPC-H query implementations, the CUDA Code-General includes an SQL parser and query planner that translate arbitrary SQL queries into the same operator trees used by the built-in queries. The parser supports `SELECT` statements with equi-joins, filter predicates, grouping, aggregation, `ORDER BY` and `LIMIT` over the eight TPC-H tables. The planner resolves table references through a catalogue, classifies `WHERE` predicates into equi-join conditions and per-table filters and determines a join order using a greedy heuristic that places the smallest dimension tables first and the largest fact table last as the probe side [2, 4, 12, 13]. The aggregation strategy is selected based on group-key cardinality: a *scalar aggregation* for queries without `GROUP BY`, a *keyed aggregation* using atomic operations on a preallocated array for small-domain keys (e.g., nation keys), or a *materialised aggregation* that writes qualifying tuples to a buffer for host-side grouping. The resulting operator tree is compiled and executed identically to the hardcoded queries through the same code-generation infrastructure.

3 DEMONSTRATION

The demonstration runs as a persistent C++ HTTP server that loads TPC-H data into GPU memory at startup and serves a single-page web application (cf. Figure 2). When the user executes a query, the server compiles and runs it in all four strategies, returning the generated CUDA source, query results and per-strategy timing in a single JSON response.

The interface is organised into six resizable panels arranged in two rows. The top-left panel contains an SQL editor with syntax highlighting, where users can either select a built-in TPC-H query (Q1–Q22) from a dropdown or write arbitrary SQL. A compiler toggle allows switching between NVRTC and nvcc. The top-right panel displays the generated CUDA source code in three side-by-side columns—one for MS, one for CG and one for DP—enabling direct structural comparison of how the same query plan maps to each strategy. Users can scroll each column independently to

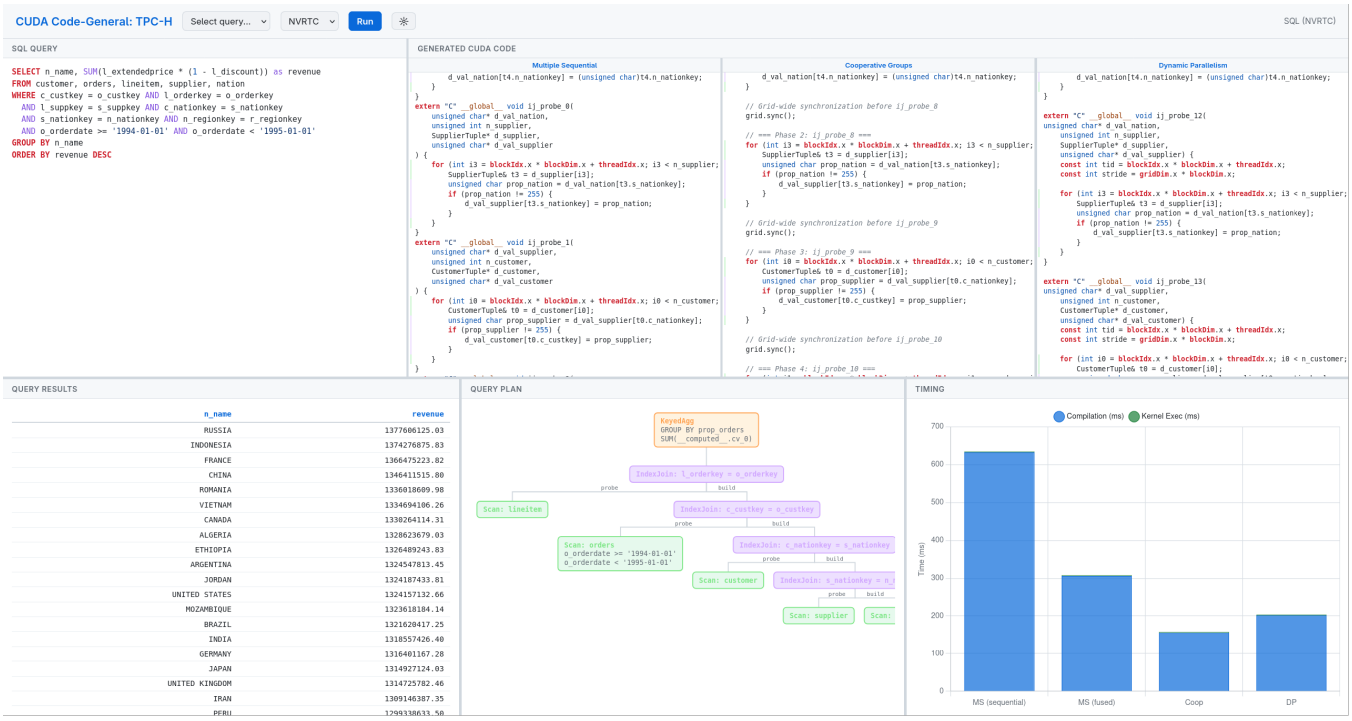

Figure 2: Screenshot of the CUDA Code-General web interface. The top row contains the SQL editor (left) and three side-by-side CUDA code columns for MS, CG and DP (right). The bottom row shows the query results (left), the query plan tree (centre) and the timing chart (right).

inspect kernel signatures, synchronisation barriers, loop structures and parameter lists. The bottom row presents three panels for analysing execution. On the left, the results panel shows the query output in a tabular format. The panel in the centre renders the operator tree, with nodes colour-coded by operator type: green for table scans, blue for semi-joins, purple for index joins and orange for aggregation operators. On the right, the timing panel shows a grouped bar chart comparing compilation time and kernel execution time across all four strategies.

Multi-Pipeline Queries and Code Structure. TPC-H Query 3 requires three pipelines: a semi-join build that marks qualifying customers in a bitmap, a second semi-join build that marks qualifying orders using the customer bitmap and a final probe pipeline that scans `lineitem`, checks both bitmaps and aggregates the discounted revenue. The MS column shows three separate `extern "C" __global__` functions, each with its own parameter list and grid configuration. The CG column shows a single kernel: the three pipeline phases are concatenated within one function body, each followed by a `grid.sync()` call that ensures all threads have completed the bitmap build before the next phase begins. The DP column shows three child kernel functions—identical to the MS kernels—plus a parent kernel that launches them sequentially with `cudaDeviceSynchronize()` between each, passing grid sizes as parameters computed by the host. MS (separate compilation) takes approximately three times longer than MS_f for the three kernels, while CG and DP compile in a single invocation and fall between the two.

NVRTC versus nvcc. The compiler dropdown allows to switch between NVRTC and nvcc. For TPC-H query 8, with nvcc, the sequential compilation takes approximately 2.9 seconds—seven separate invocations of the external `nvcc` process—while MS_f remains around 500 ms. Switching back to NVRTC, MS drops to approximately 420 ms and MS_f to approximately 91 ms. The kernel execution is identical between both compilers, confirming that NVRTC and nvcc emit equivalent PTX. The generated CUDA source in the three code columns does not change either, since the compilation backend is independent of the code-generation layer. This comparison demonstrates the practical necessity of runtime compilation for interactive query latencies: NVRTC brings the total response time well below one second.

Custom SQL Queries. Having explored the built-in queries, the interface accepts custom SQL query, for example a variant of Q5 that omits the region filter. The query plan tree now shows the planner's output: the join order, the join types (IndexJoin when a build-side value must be propagated to the probe side), the filter placement per table and the aggregation strategy—all inferred from the SQL input. The generated CUDA code is produced by the planner's operator tree rather than a hardcoded implementation, yet exhibits the same structural patterns: grid-stride loops for table scans and atomic array aggregation for the grouped `SUM`. Modifying the query—adding a filter, removing a table, changing the aggregation function—shows its effect on all panels. Removing the `nation` table, for instance, reduces the pipeline count from four to three and the compilation time drops accordingly. Changing the

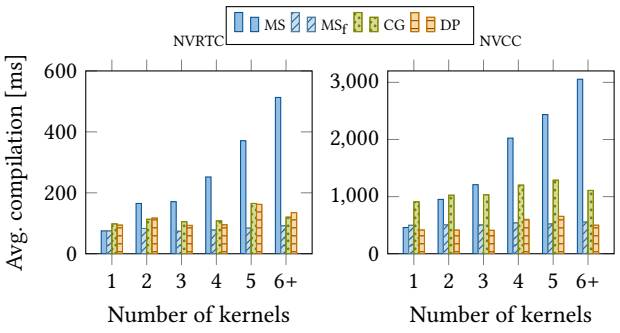

Figure 3: Average compilation time by number of kernels across all 22 TPC-H queries (SF 1) for NVRTC and NVCC.

aggregation from `SUM` to `COUNT(*)` replaces the double-precision atomic addition with an integer increment in the generated code.

Side-by-Side Code Comparison. For each query, the three code columns allow to compare how the same logical operator tree maps to three different physical execution models. In the MS column, each pipeline appears as a self-contained kernel function with `extern "C"` linkage and its own parameter list. The host launches them sequentially, each with an independently computed grid size that matches the input cardinality. Intermediate results—bitmaps for semi-joins, arrays for index joins—are allocated in global memory and persist across kernel boundaries.

In the CG column, all pipelines are merged into a single kernel function. The parameter list is the union of all per-pipeline parameters and each phase operates within a scope delimited by `grid.sync()`. The grid size is fixed to the hardware occupancy limit and every phase uses a grid-stride loop. Intermediate results in global memory are guaranteed to be visible to all threads after the synchronisation barrier, making the bitmap or array immediately available to the subsequent probe phase within the same kernel.

In the DP column, the child kernel functions are structurally identical to those in MS, but they are preceded by a parent kernel. The parent is launched by the host on a single thread and dispatches each child kernel with its own grid configuration using the Runtime API, inserting `cudaDeviceSynchronize()` between phases. Grid size variables for each child kernel are visible in the parent's parameter list, showing that the host pre-computes the per-phase grid configuration and passes it to the device.

For join operators, the build kernel writes to a bitmap (for semi-joins) or an indexed array (for index joins) in global memory using the join key as the index and the probe kernel checks its content inside an `if`-condition. In CG, both phases reside in the same kernel body, separated by `grid.sync()`. For aggregation operators, shared-memory reduction patterns—block-level partial sums accumulated with `__syncthreads()`, followed by a single `atomicAdd` per block into global memory—are visible in all strategies. In CG, the aggregation appears as the final phase of the fused kernel; in MS and DP, it is a separate kernel or child kernel, respectively.

4 EVALUATION

System: Intel(R) Xeon(R) W-2295 CPU (18 cores @ 3.00GHz, hyperthreading), 128 GB DDR4 RAM, NVIDIA RTX A2000 12 GB.
Software: Ubuntu 24.04, gcc v13.2.0, nvcc v12.0.140.
Data: TPC-H benchmark, scale factor (SF) 1

Figure 3 shows the average compilation time per strategy dependent on the number of pipelines (=kernels) for both NVRTC and NVCC. With NVRTC, MS (sequential compilation) scales approximately linearly from 75 ms for single-kernel queries to over 500 ms for queries with six or more kernels, since each kernel incurs a separate invocation. In contrast, the three fused strategies—MS_f, CG and DP—remain nearly constant at 75–135 ms regardless of the number of kernels, as they compile all kernels in a single call. The same pattern holds for NVCC at a higher baseline: sequential MS grows from 460 ms to over 3 s, while fused MS_f stays in the 500–555 ms range. For query 8 with seven kernels, fused compilation achieves a 5.9× speedup with NVRTC (576 vs. 98 ms) and 6.6× with NVCC (3752 vs. 572 ms).

5 CONCLUSION

This paper demonstrated runtime compilation and three strategies of kernel fusion for code-generating GPU database systems. An interface on top of the CUDA Code-General allowed interactive exploration of kernel fusion strategies for compiling the 22 TPC-H benchmark queries or arbitrary SQL. The demonstration made the trade-offs between fused multiple sequential kernels, cooperative groups and dynamic parallelism observable. The SQL parser and planner went beyond the benchmark, allowing ad-hoc experimentation with how join ordering, filter placement and aggregation strategy affect GPU code and execution time.

Acknowledgement. This research was supported by the German Research Foundation (DFG), grant 568456166.

REFERENCES

[1] Sebastian Breß et al. 2018. Generating custom code for efficient query execution on heterogeneous processors. *VLDB J.* 27, 6 (2018), 797–822.
[2] Konstantinos Chasialis, Yannis Foufoulas, Alkis Simitsis, and Yannis E. Ioannidis. 2026. Optimizing UDF Queries in SQL Data Engines. In *EDBT*. 247–260.
[3] Periklis Chrysogelos et al. 2019. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *VLDB* 12, 5 (2019), 544–556.
[4] Markus Dreseler et al. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *EDBT*. OpenProceedings.org, 313–324.
[5] Markus Dreseler, Martin Boissier, et al. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *VLDB* 13, 8 (2020), 1206–1220.
[6] Henning Funke and Jens Teubner. 2020. Data-Parallel Query Processing on Non-Uniform Data. *VLDB* 13, 6 (2020), 884–897.
[7] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE TKDE* 6, 1 (1994), 120–135.
[8] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *VLDB* 4, 9 (2011), 539–550.
[9] NVIDIA 2014. *NVRTC - CUDA Runtime Compilation* (7 ed.). NVIDIA. https://www.wrfranklin.org/wiki/ParallelComputingSpring2015/cuda/nvidia/doc/pdf/NVRTC_User_Guide.pdf.
[10] Johns Paul et al. 2020. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. *VLDB* 14, 2 (2020), 202–214.
[11] Anton Sachnov et al. 2024. Give a JIT on GPUs: NVRTC for Code-Generating Database Systems. In *ICDE Workshops*. IEEE, 384–387.
[12] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *SIGMOD*. ACM, 2659–2665.
[13] Xin Zhang and Ahmed Eldawy. 2024. QPJVis Demo: Quality-boost Progressive Join Query Processing System. *VLDB* 17, 12 (2024), 4345–4348.