

Tile-Based Decompression in Compiled GPU Engines

Antonio Boffa
antonio.boffa@epfl.ch
EPFL
Lausanne, Switzerland

Ananya Gupta
ananya.gupta@epfl.ch
EPFL
Lausanne, Switzerland

Hamish Nicholson
hamish.nicholson@epfl.ch
EPFL
Lausanne, Switzerland

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL
Lausanne, Switzerland

ABSTRACT

Modern GPUs reach record analytical performance only when a query's working set fits within its GPU memory budget, and that budget is under growing pressure: GPUs are partitioned and shared, working sets are outgrowing high-bandwidth memory (HBM), and HBM is a fixed on-device resource whose per-dollar capacity has stagnated across generations. Lightweight compression can relieve this pressure by shrinking both footprint and bandwidth demand, but existing GPU compressed-execution techniques either bind decompression to a specific execution model or materialize decompressed columns in HBM before query execution.

We present a compression subsystem for compiled GPU query engines that treats decompression as a first-class physical operator. Compression-specific logic is isolated behind a narrow backend interface, so new GPU-oriented encodings can be added without rewriting downstream operators.

We implement the design in Proteus and study where decompressed values should be materialized in the GPU memory hierarchy. Our compiler-integrated shared-memory strategy fuses cooperative tile decompression with tuple-at-a-time query execution inside a single generated GPU kernel, avoiding decompressed-column materialization in HBM. Compared with HeavyDB and Sirius on GPU-resident data, fused on-chip decompression achieves competitive query times while reducing the resident memory footprint by up to 4× and memory-bandwidth consumption by up to 6×.

PVLDB Reference Format:
Antonio Boffa, Ananya Gupta, Hamish Nicholson, and Anastasia Ailamaki. Tile-Based Decompression in Compiled GPU Engines. PVLDB, 20(2): 352 - 366, 2026.
doi:10.14778/3849423.3849445

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/epfl-dias/proteus/tree/decom_op.

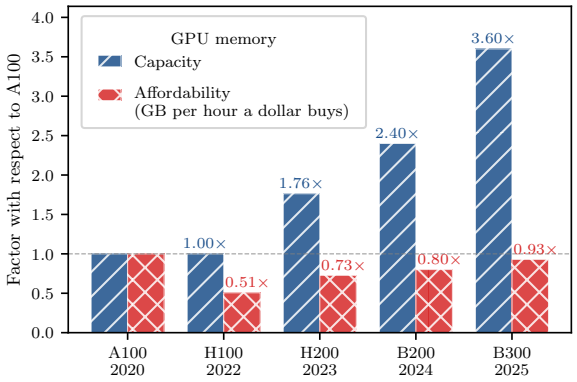


Figure 1: Affordability gap between GPU memory capacity and the memory each dollar buys. From A100 to B300, device memory grows by 3.6×, while the number of GB-hours of memory a dollar buys (averaged across four hyperscalers and one marketplace aggregator) sits below A100 levels. Capacity scales; affordability does not.

1 INTRODUCTION

Modern GPUs combine massive parallelism with multi-terabyte-per-second memory bandwidth, and recent GPU-native SQL engines show these hardware advantages can translate into record-setting end-to-end analytical performance [86, 87]. This progress builds on more than a decade of GPU database research demonstrating that large speedups for core relational operators are achieved by carefully mapping them to GPU execution and memory behavior [22, 51, 72, 80, 84, 88]. Achieving such performance, however, still depends critically on one condition: the working set must fit within the GPU memory budget.

Pressure on the GPU memory budget. This assumption is becoming increasingly fragile for several reasons. First, GPUs are increasingly shared across multiple workloads: mechanisms such as Multi-Instance GPU [14] and Multi-Process Service [15] partition a single GPU into multiple isolated instances or processes, reducing the effective memory budget available to each job [47]. Second, the working sets themselves are growing faster than high-bandwidth memory (HBM) capacity: LLM serving is bottlenecked by KV-cache size and fragmentation [56], long-context training by activation storage [92], and analytical workloads usually operate over datasets that exceed device memory [61]. Third, GPU memory is a fixed device resource: unlike CPU memory, which can often be expanded by adding DRAM, HBM capacity is determined by the GPU itself and cannot be increased without moving to a different device class or adding an entirely new GPU. Finally, memory pressure is also a cost-performance problem.

GPU memory price stagnation. Although recent hardware trends suggest that GPU memory is steadily increasing [86], this view is incomplete once cost is considered. Based on our analysis across AWS [7], Google Cloud [9], Microsoft Azure [13], Oracle Cloud [2], and marketplace aggregators [8], NVIDIA GPU device memory

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 2 ISSN 2150-8097.
doi:10.14778/3849423.3849445

capacity has grown much faster than the per-dollar memory budget, opening a widening affordability gap. As shown in Figure 1, from A100 to B300 device memory increases by 3.6× (from 80 to 288 GB), while the number of GB-hours of memory a dollar buys is still only 0.93× the A100 baseline (from 21 to 19 GB/h/$), meaning a dollar today buys slightly less GPU memory than it did five years ago, despite over three times more memory shipping on each GPU. Larger HBM pools do not imply memory is cheap in practice; GPU memory remains a valuable and expensive resource, making memory-efficient execution critical for cost-performance.

Data compression to the rescue. One approach to use as little GPU memory as possible while ensuring blazing-fast performance is lossless data compression, which reduces the volume of data that needs to be moved to and stored in precious HBM while preserving all information [46, 79, 91]. Further, as real-world datasets become increasingly redundant [26, 65, 81, 89], compression becomes even more effective.

Compression, however, is not just a space occupancy optimization: once data is kept compressed in GPU memory, it fundamentally changes how query execution accesses and manipulates values. Compressed data is not directly consumable by traditional, compression-agnostic relational operators, and in practice, there is no single "GPU compressed format": different encodings offer different compression-ratio/decompression-speed trade-offs [21, 36, 44, 60, 63, 79].

Decompress-first vs execute-on-compressed data. On one hand, systems move compressed data into the GPU and then decompress it into a fixed-width uncompressed representation for downstream operators [25, 67]. On the other hand, recent work targets executing operators directly on compressed representations, reducing both the HBM footprint and the HBM bandwidth spent reading/writing inflated values [21, 44, 46, 91].

The first line is largely driven by the host/storage-to-GPU transfer bottleneck, but GPU interconnect bandwidth is scaling rapidly (NVLink4 can reach 450 GB/s [5]). Also, materializing decompressed data into an uncompressed columnar form still incurs substantial intra-GPU overheads: it increases the HBM footprint and wastes HBM bandwidth by forcing inflated values to be written to and reread from global memory. We therefore focus on compressed execution to avoid materializing decoded columns in GPU memory, rather than only reducing the data transferred to the device.

GPU query execution models. How compressed data can be consumed depends strongly on the underlying GPU execution model. At the coarse end, operator-at-a-time engines [31, 42, 43, 86] launch one kernel per operator over an entire column, materializing intermediate results, selection vectors, validity bitmaps, or mask tensors in HBM between operators. Refining this granularity, vectorized engines process columns in fixed-size contiguous batches called tiles [78, 79]. GPU thread blocks cooperatively decode these tiles and retain cross-operator state on-chip rather than in HBM.

In contrast, compiled tuple-at-a-time engines generate pipelined GPU code in which each thread advances tuples through a fused operator sequence, aiming to minimize intermediate writes and keep values close to the consuming operators [3, 37, 53]. Proteus [34, 35, 67, 68, 76], DogQC [37, 38], Pyper [71, 72], HeavyDB [3], and Themis [45] exemplify this direction while explicitly addressing SIMT challenges such as divergence and resource contention.

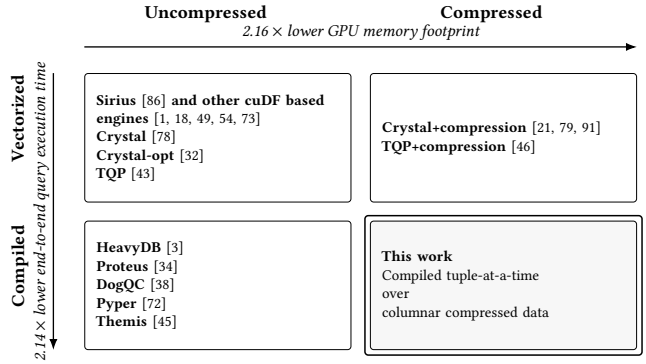


Figure 2: Taxonomy of GPU query engines along two axes: execution model (vectorized vs. compiled) and data representation (uncompressed vs. compressed). The bottom-right quadrant highlights the gap we study: compiled tuple-at-a-time execution directly over lightweight compressed data.

With these optimizations, tuple-at-a-time execution can achieve up to 2.14× shorter execution times than Crystal [78] (the fastest vectorized prototype engine [32]).

Figure 2 summarizes this landscape and motivates our focus on the unexplored quadrant: compiled GPU execution over compressed data, without eager intermediate-result materialization.

Decompression is tied to execution models. Although execute-on-compressed designs reduce HBM footprint, they typically make decompression reusable only within a specific execution model. Crystal illustrates this trade-off clearly. Its tile-at-a-time programming model makes decompression reusable within a hand-written CUDA/C++ framework, but that reuse remains at the source-code level, with fixed execution structure and hard-coded parameters [32]. It is not exposed as a compiler IR abstraction that can participate in whole-pipeline optimizations such as cross-operator inlining, explicit address-space placement, and specialization for registers versus shared memory [12, 17]. As a result, the main benefits of compressed execution have so far been realized in specialized tile-oriented systems, not in compiled tuple-at-a-time engines.

Encoding-specific operators do not scale. A second limitation is that recent execute-on-compressed designs embed format-specific decoding logic directly within relational operators (e.g., joins or aggregations) specialized for a particular encoding [46, 91]. This creates an $N \times M$ problem: supporting M encodings across N operators requires $N \times M$ specialized implementations, reproducing the usual interoperability and maintenance bottleneck [40]. This tight coupling also creates an adoption problem [40, 90]. Although new compression techniques continue to improve compression ratio and decoding speed, they are difficult to deploy when support for each representation must be hard-wired into scans, operators, or storage readers. Existing formats, therefore, tend to ossify [55]: producers keep emitting older representations to remain compatible with existing engines, while newer compression techniques remain confined to prototypes or specialized systems.

A missing boundary in the compiled pipeline. The common issue behind these limitations is the absence of a stable boundary between compressed physical layouts and relational execution. This

is the same kind of modularity problem that has pushed data systems toward composable architectures, where monolithic DBMS stacks are decomposed into reusable layers with clean interfaces (for example, language frontends [10], optimizers [23], and execution engines [57, 73]) to better adapt to changing workloads, data formats, hardware, and deployment environments [39, 48, 74]. Following this trend, we argue that compressed GPU execution also needs a reusable, compiler-visible component boundary: new compression algorithms should require new metadata-resolution and tile-decompression logic, while the rest of the compiled relational pipeline remains unchanged.

Contributions: We design, implement, and evaluate a compression subsystem for a GPU-compiled query engine that operates directly on columnar data encoded with GPU-optimized lightweight compression schemes (GPU RFOR [79] and FastLanes BitPack [21]). Prior GPU compression work shows that tile decompression can be fused into hand-written query kernels. We lift this fusion into a tuple-at-a-time GPU query compiler by making decompression a first-class physical operator: compressed columns are decoded into the same tuple-binding interface used by uncompressed scans, allowing existing relational operators to consume them unchanged. This provides compiler-mediated fusion of GPU decompression and relational execution while preserving hardware-aware placement choices for decoded values. Our contributions are:

(i) We introduce `CompressedBlockToTuples` (`CB2T`), a physical data-flow decompression operator for compiled GPU query engines. It exposes columnar-oriented compressed columns as tuples, isolating compression-specific metadata resolution and tile loading behind a backend interface while preserving the relational operator interface (Section 3).

(ii) `CB2T` enables compiler-mediated fusion of tile decompression with tuple-at-a-time GPU query pipelines. The generated code passes decoded values to downstream operators through global-memory, shared-memory, or register-backed bindings while preserving the standard tuple-at-a-time interface (Section 4).

(iii) We integrate `CB2T` into Proteus and evaluate the runtime, memory traffic, and device-memory footprint trade-offs of uncompressed execution, eager decompression, staged global-memory, and fused on-chip decompression. Fusing decompression on-chip strictly dominates, delivering faster, smaller results on every query, while against uncompressed execution it roughly halves the resident footprint at a modest, query-dependent cost. Profiling further shows the best decode placement and launch geometry are query-dependent, and that peak occupancy does not always minimize runtime. Across SSB [70], TPC-H [19], and Stack [64] queries, we place these results in the broader GPU-SQL landscape, by comparing Proteus with `CB2T` against two state-of-the-art production GPU engines: HeavyDB [3], a compiled tuple-at-a-time engine, and Sirius [86], a recent libcudf-based engine. Proteus with fused on-chip decompression achieves competitive query times while reducing the resident GPU memory footprint by up to 4× and GPU memory-bandwidth consumption by up to 6× (Section 5).

2 PRELIMINARIES AND RELATED WORKS

2.1 GPU architecture essentials for database kernels

A GPU[1] has many streaming multiprocessors (SMs). Threads are executed in groups of 32 called warps, and warps are scheduled on SMs. A thread block is a group of warps and is the basic unit of thread cooperation, synchronization, and shared-memory allocation. Each thread has a private register file, which is the fastest but most limited storage level. Part of the on-chip memory in an SM can be configured as shared memory, explicitly managed by the programmer and visible to all threads in the same thread block. This allows frequently reused data to be staged on chip and reduces accesses to off-chip global memory. In CUDA, a kernel's execution configuration is fixed at launch time through `blockDim` (threads per block) and `gridDim` (blocks per grid). All thread blocks in the kernel share this configuration, and therefore all threads execute under the same launch parameters. The GPU can place multiple thread blocks on the same SM, but only while respecting hardware limits. Increasing shared memory per block or registers per thread can therefore reduce the number of resident blocks and warps per SM, lowering occupancy and reducing the ability to hide latency. Register pressure has two distinct effects. If the compiler cannot allocate all live values to registers, it emits spill stores and spill loads to thread-local memory. Although this memory is private to a thread, it is backed by the device memory hierarchy and is much slower than registers.

2.2 Compilation-based query execution and the GPU memory hierarchy

Compilation-based engines [3, 52] fuse relational pipelines by translating operator interactions into direct dataflow inside generated code, rather than materializing intermediates between operators. This idea follows the data-centric, produce/consume compilation model introduced by HyPer [66], where values are kept in registers as long as possible. In this model, values loaded by a scan flow directly into predicates, hash computations, and aggregations within one fused code fragment. LLVM provides a single intermediate representation that can be lowered to different targets (e.g., x86 for CPUs and NVPTX for NVIDIA GPUs).

A scan operator can avoid exposing tuples through a generic runtime interface; instead, it can rely on input plug-ins [52] that generate the access code needed for the specific storage format and place the required fields into virtual buffers. These buffers act as a tuple-level intermediate representation for the rest of the generated pipeline, and LLVM can often promote them to registers.

From block-oriented storage to tuple: `BlockToTuples`. We call `BlockToTuples` (`B2T`) the component that bridges the uncompressed block-oriented storage layer and its tuple-oriented execution model: it iterates over the logical tuple positions in a block, exposing each as the tuple identifier and the requested attributes as ordinary tuple bindings. For each position and field, it computes the element address with an LLVM `getelementptr` on the block base pointer and current index (issuing no memory access) and stores it in an

[1] We focus on NVIDIA GPUs. Other architectures, such as AMD GPUs, follow similar principles.

Table 1: Where compact or compressed representations enter GPU query execution. The distinction is whether compression is hidden behind a scan/decode boundary or exposed to downstream operator logic.

System / class	Execution model	Compression interface	Operator contract	Compression coupling
Sirius [86]; cuDF/Arrow engines	Vector/op.	Decode outside pipeline	Operators consume decoded arrays	Outside pipeline
Proteus [34, 35, 76]; DogQC [38]; Pyper [72]; Themis [45]	Tuple/JIT	Uncompressed scan/B2T	Operators consume tuple bindings	No compressed boundary
HeavyDB fixed-width storage [3]	Tuple/JIT	Fixed-width column load	Operators consume tuple bindings	Type-narrowing load path
Crystal [78]; Crystal-opt [32]	Tile/AOT	Uncompressed tile load	Tile kernels consume staged values	No compressed boundary
GPU-FOR/RFOR [79]; FastLanes-GPU [21]	Crystal tile/vector	Load/decode routine	Tile kernels consume decoded values	Codec-specific load path
GTileQuery [91]	Crystal tile	Direct compressed operators	Operators inspect compressed metadata/values	Operator–codec coupling
TQP [43]	Tensor/column	Dense tensor representations	Tensor operators consume dense tensors	No compressed boundary
Compressed TQP [46]	Tensor/column	Encoded tensor representations	Operators rewritten over RLE/Index/Plain tensor layouts	Encoding-aware operators
CB2T in Proteus	**Tuple/JIT**	**Compressed B2T physical operator**	**Operators consume ordinary tuple bindings**	**Backends only**

`alloca` slot. LLVM's `mem2reg` pass then promotes these slots to SSA register values, leaving the per-element load emitted by the consuming operator as the only surviving memory access [12]. The identical IR is lowered to GPUs by the NVPTX backend without rewriting operator logic: the loop stride is set to the total thread count, so each thread processes a strided subset of positions, matching the SIMT model. For GPU targets, LLVM IR follows NVVM address-space conventions [17], where pointer types distinguish generic, global, shared, and local memory; these address spaces are preserved by NVPTX lowering and determine the corresponding PTX load/store instructions.

2.3 GPU database execution models

Table 1 organizes the GPU query engines discussed in this section by their execution model and by where compression meets operator logic, and situates `CB2T` among them. We group these engines into three execution-model classes: tuple-at-a-time, vectorized, and tensor-at-a-time.

Tuple-at-a-time execution. A first line of work adopts a compilation-based, tuple-at-a-time model, where multiple relational operators are fused into a small number of query-specific GPU kernels.

Proteus is a GPU-accelerated columnar database that treats input formats through plug-ins and dynamically generates query-specific execution code, following a push-based, data-centric compilation model in which relational pipelines are fused into a single kernel (using LLVM [59] and PTX [16]). Funke et al. [37] introduced fusion operators for GPUs, generating fused OpenCL/CUDA kernels that process tuples through data-parallel nested-loop pipelines. Subsequent work refined this model to match GPU execution constraints. He et al. [38] proposed a JIT-compiled GPU query engine that mitigates warp divergence through lane refill and push-down operators, while Paul et al. [72] introduced the Shuffle and Segment transforms to reduce divergence and resource contention in compiled GPU pipelines. More recent work improves load balance by dynamically redistributing work across threads and warps [45]. Optimizing warp utilization is orthogonal to our work; we integrate decompression into the same fused kernels via LLVM/NVVM.

HeavyDB [3] (formerly OmniSci, formerly MapD), like Proteus, JIT-compiles fused query pipelines to GPU code through LLVM and PTX. HeavyDB stores columns as contiguous arrays of fixed-width, byte-aligned values: the address of row i is computed directly from i and the column's physical type width. This makes positional access simple and GPU-friendly, but it also means that HeavyDB does not use variable-length lightweight encodings such as RLE. The only data representation it allows is therefore fixed-width integer narrowing, e.g., choosing an 8-, 16-, or 32-bit integer type for an entire column when the column range permits it.

We use HeavyDB in our experiments as a production-grade exemplar of the compiled tuple-at-a-time category, an independent cross-system baseline alongside Proteus; its restriction to inlined, fixed-width column loads isolates the rigidity that `CB2T` avoids by compressing at tile granularity.

Vectorized execution. A second line of work follows a vectorized processing model. On CPUs, this typically means that each operator processes a vector of columnar values at a time [30]. On GPUs, the same idea naturally extends to a tile-at-a-time model [78], where a thread block cooperatively loads and processes a fixed-size tile in shared memory. Because a thread block loads cooperatively into shared memory and consumes the result within the same block, the tile-at-a-time model is where lightweight GPU compression has mostly been integrated, which we discuss in Section 2.4. We omit Crystal-derived engines in our cross-system comparison: Crystal is not a SQL-driven system, since each SSB query is implemented as a hand-written CUDA kernel with hard-coded hash-table sizes, group counts, and output cardinalities, though it remains the reference for tile-at-a-time GPU execution. We discuss the benefit of `CB2T` with respect to Crystal-derived engines in Section 5.7.

Cao et al. [32] analyze Crystal and other GPU DBMSs using roofline modeling and microarchitectural profiling, showing that kernel fusion, thread efficiency, and memory-system utilization remain central bottlenecks even in highly optimized GPU engines.

Sirius [86] adopts a GPU-native, vectorized execution model built on top of libcudf. While it provides modular, drop-in acceleration via the Substrait plan format, its reliance on an uncompressed, Arrow-derived columnar format results in high HBM bandwidth consumption. Sirius treats decompression as an external pre-processing step and materializes uncompressed intermediate blocks between operator calls, missing the opportunity for the fused, compressed execution we propose. We test Sirius in our experiments as a state-of-the-art exemplification of the libcudf-based category.

Tensor-at-a-time execution. A third direction is to move GPU query execution into tensor runtimes. The Tensor Query Processor (TQP) [43] compiles SQL into tensor programs executed by frameworks such as PyTorch, TVM, or ONNX Runtime, effectively treating relational pipelines as tensor computations scheduled by the ML stack. Conceptually, this is a batched execution model closer to tile-based processing than to compilation-based tuple-at-a-time

engines. We do not benchmark TQP in our experiments: prior cross-system measurements [32] report TQP at roughly 4× HeavyDB's GPU-compute time on SSB, so the HeavyDB baseline already dominates TQP on the GPU-internal dimension our work targets.

2.4 Lightweight GPU integer compression

Lightweight compression schemes, e.g, run-length, dictionary [24], delta, bitmaps [33], FSST [29, 83], XOR [75], Null suppression [77], and learned schemes [27, 41, 62] operate over blocks of column values [20]. They have been shown to be more effective than heavyweight compression schemes on GPUs [67]. They are attractive for GPU analytics because they reduce data movement while keeping decoding simple enough to run close to query execution. GPU decompression is not only about choosing an encoding with a good compression ratio and/or decompression speed; the format must also expose sufficient parallelism for thousands of threads.

Early GPU compression schemes optimized individual lightweight primitives, such as null suppression and bit-packing [63]. More recent designs move toward tile-based cascades, where multiple primitives are combined [36, 60]. The primitives most relevant to this work are: bit-packing (BP), which stores each value using as few bits as possible, just enough to represent the column's value range; Frame-of-Reference (FOR), which subtracts a per-block minimum and bit-packs the residuals; and run-length encoding (RLE), which replaces a run of repeated values with a $(value, rl)$ pair. Most GPU systems that execute over lightweight encodings extend the vectorized, tile-at-a-time model of Section 2.3, and differ mainly in their encodings and in how tightly they couple decoding to operator logic. Shanbhag et al. [79] introduce GPU-RFOR, a compression-ratio-oriented scheme that cascades RLE, FOR, and BP: it encodes each tile as two streams (values and run lengths), each compressed with FOR. The cascaded design achieves higher compression ratios than single-layer encodings, but the multi-layer decode path is more compute-intensive. Its reference implementation is open source and exposes the decoder as GPU device code.

FastLanes-GPU [21] optimizes for decode throughput. Its FastLanes BP (FLBP) scheme applies FOR followed by BP in an interleaved layout that distributes values round-robin over SIMT lanes, so each thread in a warp decodes its share independently, without inter-lane transfers, yielding high bit-unpacking throughput. The trade-off is a lower compression ratio, since FLBP does not cascade RLE or other encodings (which have not been ported to CUDA).

Other tile-at-a-time designs push decoding past the codec and into operator logic itself. Zhang et al. [91] (GTileQuery) embed encoding-specific decoding directly within relational operators, so decoding runs as part of operator evaluation. Operating on the compressed representation directly avoids materializing decompressed columns in global memory and lowers memory traffic. The trade-off is generality: each operator is specialized to its encoding, coupling the storage format to the operator implementation.

Building on the tensor-at-a-time model (section 2.3), Huang et al. [46] extend TQP with tensor-level primitives for filters, joins, and aggregations that operate directly on lightweight compressed formats. This keeps data compressed throughout execution but again intertwines physical encodings with runtime-specific operator implementations. Our goal is neither to design another GPU integer encoding nor another encoding-specialized relational operator.

2.5 Challenges of compiled execution over compressed columns

A first challenge is a structural mismatch: lightweight encoding schemes align well with vectorized execution. A tile corresponds to a block of columnar values. Decompression can be implemented as a block-wide primitive applied to that tile. This explains why prior work integrates compression naturally into vectorized GPU engines [79]. Tuple-oriented, compiled execution, in contrast, operates on tuples via a fused pipeline [66].

This breaks the sequential, block-local access patterns that make column encodings efficient, increasing control divergence (variable-length codes, RLE runs, dictionary lookups), register pressure (multiple decode cursors/state), and memory traffic due to scattered reads across columns.

A second challenge follows from this. Since decoding one value at a time wastes the work shared across a tile, compressed columns are decoded cooperatively, a tile at a time. Their values must then be staged somewhere in the GPU memory hierarchy (registers, shared, or global memory) before the tuple-at-a-time pipeline reads them. Unlike CPUs, where much of the cache hierarchy is transparent to the engine, GPU memory placement is explicit in the generated code: LLVM IR targeting NVPTX follows NVVM address-space conventions, where shared and global memory are distinguished through pointer types that lower to different PTX load/store instructions. The staging-location decision must therefore be exposed to address-space handling and register allocation, yet hidden from the operator interface. Otherwise, every operator would need to know how that column was materialized.

3 THE MODULAR DECOMPRESSION OPERATOR: A CONTRACT FOR COMPOSABLE COMPRESSED SCANS

The design goal of `CompressedBlockToTuples` is to keep input data compressed in a columnar layout while feeding it to tuple-at-a-time relational operators within a compiled GPU query engine, without spilling decoded intermediates back to HBM. Doing so lowers the effective per-GB execution cost while keeping decompression composable. This section specifies the design (the contract `CB2T` commits to and why that boundary is the right one), and Section 4 realizes it as a JIT-compiled, fused GPU kernel.

Table 1 summarizes how prior GPU query engines handle compressed data (detailed in Section 2): they are decompressed outside the pipeline into materialized intermediates, restricted to fixed-width type narrowing, or fused with execution but only by embedding compression-algorithm-specific logic inside relational or tile operators. `CB2T` is the only point in this space that stays tuple-at-a-time and JIT-compiled while confining all compression-algorithm knowledge to a reusable backend, so downstream operators consume ordinary tuple bindings regardless of the encoding, which is what we mean by a composable decompression operator.

Composability is well-understood for traditional relational operators and other engine components, but what composability means

for decompression inside a compiled GPU query engine has not been characterized. We decompose composability for data compression into two concrete sub-properties:
(i) **Interface substitutability.** A compressed column presents the same logical attribute interface as an uncompressed column: relational operators consume values through the engine's standard scan/tuple interface rather than through compression-algorithm-specific APIs (Section 3.1).
(ii) **Decompression encapsulation.** Compressed representation details, such as metadata layout, bit packing, run structure, and tile decoding, are confined to a reusable backend layer and do not appear in relational operator implementations. Therefore, the backend is responsible for resolving compressed-tile metadata and producing decoded tile values through the common interface; downstream operators remain oblivious to whether the values originated from raw memory, RFOR, bit-packing, run-length encoding, or another tile-decodable format (Section 3.2).

3.1 Interface substitutability

Figure 3 summarizes the operator structure. From left to right, the operator receives compressed blocks that have already been moved to the GPU. A block is the unit of transfer and storage; internally it is partitioned into fixed-size tiles (the granularity at which the operator cooperatively decodes values and feeds them to the pipeline). `CB2T` sits inside the fused query processing kernel and exposes two format-specific hooks, `resolve_metadata()` and `decompress_tile()`, which are the subject of Section 3.2. In the figure's bottom-right corner, the orthogonal execution knobs control decode placement and launch configuration; these knobs are the levers exploited by Section 4.3. Everything compression-algorithm-specific lives behind those two hooks; everything downstream of the operator boundary is fixed across compressed formats.

`CB2T` exposes a tuple-facing interface (the right-side arrow leaving the operator in Figure 3, labeled standard tuple interface for next operators) identical to the one a regular uncompressed scan would produce. Filter predicates, projections, hash-probes, and aggregations execute against ordinary tuple bindings and have no awareness of whether the values originated from a compressed or uncompressed source; their generated code is identical.

3.2 Decompression encapsulation

All compression-specific knowledge is confined to the backend layer shown in Figure 3 as the Pool of compression backends, which contributes only two hooks: `resolve_metadata()`, which interprets the per-tile layout, and `decompress_tile()`, which decodes one tile of values according to that layout. The two hooks follow the asymmetric cooperation model that is characteristic of lightweight GPU decompression: `resolve_metadata()` runs on a single block leader (one thread per block), which parses the per-tile header and prepares the fields, while `decompress_tile()` executes cooperatively across all threads in the block, reading the metadata and emitting one decoded tile per column.

This narrow backend contract also makes compressed format integration low-effort: adding a new one requires implementing only the two backend hooks for that format, how to interpret its metadata, and how to decompress one tile. This contract targets

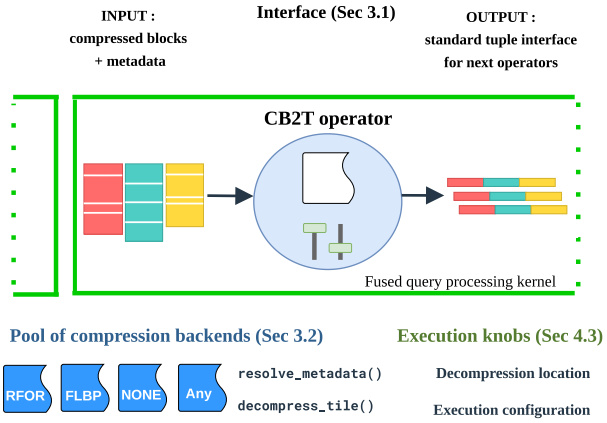


Figure 3: The modular decompression operator inside a fused query kernel: it consumes compressed tiles with per-tile metadata and emits a stable tuple stream to downstream operators (top), draws on interchangeable backends that implement only `resolve_metadata()` and `decompress_tile()` (bottom-left), and is tuned by decode-placement and launch-configuration knobs (bottom-right).

lightweight, tile-decodable column encodings, whose decode reduces to per-tile metadata resolution plus cooperative tile materialization [21, 79, 85, 91].

These hooks can be provided either as CUDA device functions or through the engine's LLVM IR generation interface, depending on whether the compressed format is implemented as hand-written GPU code or generated during query compilation. Once these two functions are registered, the new format automatically inherits the same logical column interface as existing encodings. Compression algorithms are therefore free to evolve (new tile parameters, new header fields, even new variable-size sections) without touching any other component of the engine.

Self-describing blocks for engine-side isolation. Compression-related block metadata has to reach the decoder, but if it lived in tables maintained by the engine, it would force every component that touches a column to carry compression-aware state. `CB2T` eliminates this coupling by colocating all block metadata with the data it describes: each compressed block opens with a fixed header (format identifier, tile count, per-tile offsets), followed by the compressed payload (a sequence of tiles, each prefixed with its own metadata describing how to decode it). At scan time, the engine hands `CB2T` block pointers and nothing else. Nothing outside `CB2T` ever inspects compression-specific state: the rest of the engine treats compressed blocks as opaque byte ranges, indistinguishable from uncompressed ones, and routes them through the existing block-oriented path (allocation, transfer, eviction, mem-move) unchanged.

`CB2T` vs. data-access specialization. Input plug-ins [52] specialize in how values are accessed from concrete dataset representations such as CSV, JSON, or relational binary layouts. They hide representation-specific details from operators by generating code that locates, parses, converts, or lazily reconstructs the values required by the query.

CB2T operates one layer later. It defines how that block is expanded into tuple bindings inside the fused pipeline. This is an execution problem rather than only an access problem: lightweight GPU decompression requires tile-level cooperation, metadata resolution, synchronization, and explicit placement of decoded values in shared memory or registers. In this sense, CB2T is a hardware-software co-designed component: it exposes the GPU structure needed for efficient decompression while keeping compression support composable.

4 REALIZING THE CONTRACT: A JIT-COMPILED, FUSED GPU KERNEL

Section 3 defines the operator interface and the format-specific backend contract that CB2T commits to at every site in the plan. This section shows how that contract is implemented inside a compiled GPU engine: each thread block first cooperatively decompresses a tile-sized row group of every compressed column into on-chip storage, then immediately invokes the next operator's tuple-at-a-time `consume()` code over the decoded tile, with both phases living in the same compiled kernel. Compilation is not part of the contract; it is the enabler that makes this implementation feasible. Because the operator boundary is represented inside the same LLVM IR module as the rest of the query, and lowered through the NVPTX backend following NVVM address-space conventions, the cooperative tile decode and the per-thread tuple pipeline can be fused into a single compiled kernel. The optimizer can then inline, propagate values, and eliminate redundant materialization across the decompression/operator boundary.

As Section 5.5 later shows, fused shared-memory decompression is the fastest compressed placement in every evaluated query–codec case. We therefore focus this section on the non-trivial system design choices required to realize the contract from Section 3 efficiently. Section 4.1 describes how we use shared-memory as a scratchpad, bridging cooperative decode and tuple-at-a-time execution; Section 4.2 shows how the tuple binding handed to the next operator stays substitutable across the JIT boundary; Section 4.3 unpacks the execution knobs that govern decode placement and thread/tile configuration.

4.1 Shared-memory as the bridge

The mismatch between compressed input and tuple-at-a-time execution arises from mixing two access patterns: lightweight compression algorithms decode columnar values block-locally and cooperatively, with the entire thread block decoding as a single coordinated operation, exploiting the encoding's spatial locality and uniform structure. A tuple-at-a-time pipeline accesses values row-wise, in which each thread executes a fused sequence of operators.

Shared memory bridges the access pattern. Our operator emits a two-level loop: an outer loop assigns compressed tiles to thread blocks and invokes a cooperative decoder (encapsulated in the two hooks described in Section 3.2) that writes one tile per column into shared memory (using it as scratchpad); the inner tuple loop has each thread read its assigned values from the scratchpad, populate the tuple environment, and inline the `consume()` code of every downstream operator. Two `__syncthreads()` barriers bracket the inner loop (one ensures the tile is fully decoded before any thread reads it, the other prevents overwrite while another thread is still consuming). Both phases live in the same compiled kernel, so decoded values flow from the decoder into the tuple loop through the shared scratchpad, without ever materializing in HBM.

4.2 Preserving the tuple binding across the JIT boundary

As discussed in Section 2.2, B2T works because the data is already resident, uncompressed in global memory, and can be read directly. CB2T has no such option because it deals with compressed, variable-sized tiles and it decodes them in a cooperative, block-wide computation that must first reconstruct values and place them somewhere in the GPU memory hierarchy before the tuple-at-a-time pipeline can read them; registers, thread-block shared memory, and global memory each carry different costs in bandwidth, capacity, and occupancy, and the choice is observable in the IR because Section 2.2 reminded us that the NVPTX backend treats pointers in different address spaces as distinct types. Interface substitutability (Section 3.1) therefore demands that this placement decision terminate inside CB2T: the binding handed to the next operator must look identical to the one produced by B2T, regardless of where in the memory hierarchy the underlying value lives.

CB2T meets this requirement with a two-sided interface around shared memory. Inside the operator, the cooperative decode routine is declared with shared-typed parameters and writes each column's decoded tile directly into shared memory. Externally, the same per-column scratchpad slot is exposed as a generic pointer, and that pointer is what enters the tuple binding consumed by the next operator. Downstream code issues a vanilla load on the generic pointer, unaware of where the value resides, and without a rewrite.

This encapsulation adds no indirection. Although the binding is exposed as a generic pointer, the backend can infer that it refers to the shared memory and emits a direct shared-memory access instead of a runtime-converted generic one, so downstream operators read each attribute directly from the on-chip tile. We confirmed this on the emitted PTX: shared memory accesses lower to `ld.shared`/`st.shared` with no address-space conversions, and the only remaining global loads are of the compressed input.

4.3 Execution knobs: decode placement and launch configuration

The shared-memory scratchpad of Section 4.1 is where CB2T places decoded values by default, but shared memory is not the only option. CB2T exposes two execution knobs that vary its physical realization without changing the external tuple binding:

(i) **Decode placement.** We treat decode placement (where decoded values live between the cooperative decoder and the tuple-at-a-time loop) as an explicit design axis with three points: global memory, shared memory (Section 4.1), and registers (Section 5.3). The latter variant reduces shared-memory pressure by keeping only one decoded column tile in shared memory at a time and lifting each thread's assigned values into registers before reusing the scratchpad slot for the next column. These variants preserve the same external tuple binding; they differ only in where CB2T terminates the decode internally. For comparison, Section 5.3 also evaluates a global-memory materialization baseline, where decompression

is performed by a separate kernel into an HBM buffer and the unchanged query pipeline later reads the decoded values.

(ii) **Launch configuration.** The launch configuration determines how a fixed compressed tile is mapped to CUDA threads during decode and tuple consumption. The tile size, T, is the number of values handled per tile. This is a physical format parameter fixed when the dataset is compressed. At execution time, CB2T spreads the tile over a thread block: `threads_per_block` (`tpb`) threads each decoding `items_per_thread` (`ipt`) values, where $T = \texttt{tpb} \times \texttt{ipt}$.

Occupancy as a resource tradeoff, not a bottleneck. Per-column shared tiles introduce a resource constraint absent from uncompressed scans. A fused compressed scan must reserve space for decoded values, metadata, and any compression-method-specific auxiliary state before the tuple-at-a-time loop consumes the tile. For K columns and tile cardinality T, the decoded-value footprint alone scales with $K \times T$. This staging footprint can limit occupancy by reducing how many thread blocks fit concurrently on an SM. CB2T therefore chooses occupancy as a tunable resource tradeoff rather than a fixed bottleneck.

The first decision, the tile size T, is made when the data is compressed rather than at execution time. GPUs with larger shared-memory budgets can often afford larger tiles, while smaller shared-memory budgets or wider projections favor smaller tiles.

The second decision, made once T is fixed by the compressed layout, is how to map the tile onto a thread block; CB2T exposes it through `tpb` and `ipt`. Larger `tpb` (smaller `ipt`) increases the number of resident warps per block and lowers per-thread register pressure, at the cost of more synchronization and setup overhead amortized over fewer values per thread; smaller `tpb` (larger `ipt`) does the reverse. Which end of this tradeoff is preferable is query- and codec-dependent and cannot be explained by occupancy alone, Section 5.5 quantifies it empirically.

Exposing the configuration matters because decode and query share one kernel. Because CB2T and the downstream relational operators execute inside the same GPU kernel, the decoder's cooperative shape is also the shape under which the fused query pipeline runs, while the query side competes for the remaining registers and cache. Joins and aggregations add probe and aggregation state on top of the shared memory already held for decoded tiles, so the configuration that maximizes decoder occupancy is rarely the one that minimizes end-to-end time. This is why CB2T exposes `tpb` and `ipt` as first-class knobs instead of hard-coding them: the same decoder can be retuned per query to balance its cooperative access pattern against the fused pipeline's footprint, without touching the decode routine or any relational operator. Section 5.5 quantifies this query-dependent sensitivity.

5 SYSTEM INTEGRATION AND EVALUATION

We implement our CB2T and integrate efficient decompression into a composable, compilation-based, tuple-at-a-time GPU query engine.We use Proteus as an example, which has the right infrastructure to study how decompression at a specific stage of query execution can affect query execution time [67].

Proteus explicitly abstracts the storage-to-pipeline transition through the B2T operator (Section 2.2). This allows us to implement CB2T (Section 3) as a first-class, modular operator, capable of handling complex address-space transitions and collaborative decoding logic without modifying downstream relational operators. In this section, we answer three research questions:

RQ1 What is the end-to-end GPU memory-traffic cost of decompressing into a temporary GPU buffer prior to query execution, versus fusing it into the query pipeline? (Section 5.4)

RQ2 Given this traffic analysis, what is the end-to-end query execution time of on-chip decompression, and is it always beneficial to fuse decompression and query execution given its shared-memory and register pressure? (Section 5.5)

RQ3 How does fused on-chip decompression in Proteus compare in end-to-end runtime and GPU-memory footprint against state-of-the-art GPU engines: HeavyDB and Sirius? (Section 5.6)

Hardware. We ran the complete evaluation on both a local NVIDIA A40 GPU with 48 GB of GDDR6 memory and a cloud NVIDIA A100-SXM4 GPU with 40 GB of HBM2 memory, obtaining consistent qualitative conclusions and similar relative performance trends. We report the A100 results because its HBM memory system more closely matches the hardware setting motivating this work (Section 1). The A100 instance was provided by lambda.ai [11, 86] and runs Ubuntu 22.04.5 with Linux kernel 6.8.0. The instance exposes 30 CPU cores of an AMD EPYC 7J13 processor and 216 GiB of DRAM. Each system is built and run with its system-specific toolchain. Proteus is built with CUDA 11.3 and LLVM/Clang 14.0. HeavyDB was built with CUDA 11.8, GCC 12.4, and LLVM/Clang 14.0. Sirius uses DuckDB v1.5.2, CUDA 12.9, and GCC 14.3.

Dataset. We use the SSB dataset SF 100, selected TPC-H queries SF 100, and real-world Stack queries over the fixed StackExchange dump. Following standard practice in GPU databases [78, 79], for string columns, regardless of the compression scheme used, we apply dictionary encoding during data preparation, uniformly across all tested systems. Each unique string value is mapped to a 4-byte integer ID using a static ordered dictionary constructed from the full column. Each column is partitioned into fixed-size 2MiB blocks: the granularity Proteus uses to overlap host-to-device transfer with processing; our in-GPU-resident experiments preload all blocks and exclude transfer, but keep this block size for fidelity with the general data path [25, 67] (see Section 5.1). In the compressed dataset, each block is further divided into 1024-value tiles. In Section 5.5, we decrease RFOR tile size to T=512 since it is more tile-size–sensitive than FLBP because each tile stages more shared memory (run-lengths and cooperative scan buffers).

Workload. We run the SSB queries Q1.1/2.1/3.1 that scan 9.6 GB and Q4.1 that scans 14.4 GB per query. We additionally report TPC-H Q5 and Q12, whose uncompressed referenced-column footprints are 11.5 and 13.2 GB, respectively. Both join `lineitem` and `orders`, the two largest TPC-H relations. Finally, among the Stack workload, we select Q11 as a representative multiway join and Q14 as the deepest join plan, spanning four to seven relations and three to six hash joins over real-world data. Cross-system results on these workloads should be interpreted as production-baseline comparisons: the systems differ in memory-pool policy, type lowering, SQL rewriting, and join planning. Building on studies that characterize GPU query engines across systems and hardware [32, 50] and on single-system comparisons of competing execution designs [53], future work will

Table 2: Compression ratio and aggregate GPU decompression throughput (GB/s) over each query's columns.

	FLBP		RFOR		nvCOMP Casc.	
Query	Ratio	GB/s	Ratio	GB/s	Ratio	GB/s
SSB Q1.1	2.45	**713**	**2.96**	301	2.64	135
SSB Q2.1	1.59	**576**	**1.82**	282	1.80	138
SSB Q3.1	1.52	**559**	**2.30**	296	2.26	160
SSB Q4.1	1.54	**566**	**2.02**	286	2.01	145
TPC-H Q5	2.08	**653**	2.19	284	**2.25**	136
TPC-H Q12	2.57	**729**	**2.74**	271	2.58	131
Stack Q11	3.54	**625**	**4.18**	313	4.12	125
Stack Q14	3.19	**646**	3.38	315	**3.49**	131
Overall	2.03	**647**	**2.24**	280	2.15	129

compare across more workloads, execution paradigms, hardware, and data representations.

5.1 In-GPU evaluation methodology

Proteus was designed for heterogeneous CPU-GPU execution. It moves data as per-column 2MiB blocks; once a row group's blocks are all resident, it launches the compiled pipeline over the row group, preserves state across launches, and continues to the next.

Here we switch to a single-kernel-per-pipeline execution mode: each compiled pipeline runs as a single kernel launch over the entire input (as in HeavyDB). All input data are already resident in GPU global memory, and this mode lets us isolate the impact of decompression location within the GPU memory hierarchy. On the uncompressed baseline, single-kernel-per-pipeline reduces query time by 19% on average, demonstrating that for in-GPU execution, the per-launch CPU-GPU control-plane cost is itself a first-order term once data movement is removed.

5.2 Compression algorithms

In all experiments, the same compression algorithm is applied to every column and block. Strategies that select different compression algorithms per column or per block (potentially improving compression ratios by matching the encoding to each column's value distribution [28]) are left for future work, along with an analysis of the additional query compilation time [58].

Inside the `CB2T` operator, we evaluate two open-source, lightweight integer compression schemes: RFOR and FLBP (introduced in Section 2.4), which represent different points in the design space (as shown in Table 2): one optimized for compression ratio via cascaded encoding layers, and one optimized for raw decompression speed via a simpler, fully data-parallel layout.

We define the compression ratio as the ratio of the uncompressed representation to the compressed representation. We report compression ratios and decompression speed for NVIDIA nvCOMP Cascaded [4] as a reference point, but do not integrate it as a query operator: nvCOMP is a closed-source library whose decompression routines are not available as GPU device code, making it impossible to inline into the LLVM IR compilation stack and fuse with downstream operators within the same kernel; furthermore, it is outperformed by schemes with fused, single-kernel decoders [21, 44, 79].

Table 2 reports compression ratio and decompression throughput per query, aggregated over the columns each query references.

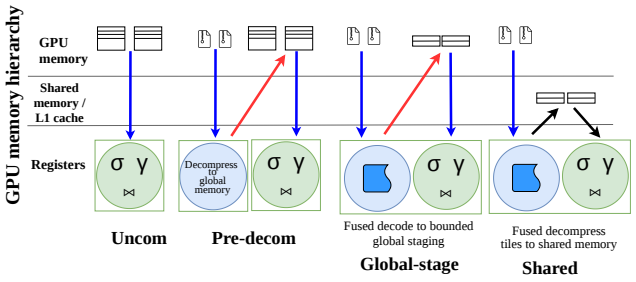


Figure 4: Decompression placement strategies for GPU query execution. Arrow colors correspond to the segments in Figure 5: blue = HBM read, red = HBM write.

Throughout this section, every measurement averages 10 timed runs after one discarded warmup.

The two schemes trade ratio for speed. RFOR compresses best on six of eight query families and overall (2.24× vs. 2.03× for FLBP and 2.15× for nvCOMP Cascaded), as RLE captures long runs, notably in `lo_orderdate` and `lo_custkey` for SSB Q1.1/Q3.1. FLBP cannot exploit these runs, but its warp-level, synchronization-free decoder is fastest on all eight families (559–729 GB/s).

5.3 Decompression strategies

We evaluate the placements in Figure 4:

(Uncom) Uncompressed baseline. In the baseline, the data is stored uncompressed in GPU memory. The `B2T` operator has each thread load the tuples it needs and feed the rest of the pipeline.

(Pre-decom) Decompress to global memory. Compressed blocks are decompressed into a preallocated global buffer, after which the unchanged baseline pipeline reads tuples from that buffer. The implementation launches a kernel that performs a cooperative decode using dynamic shared memory; each block writes its decoded tile directly into a preallocated global memory output buffer. Downstream code is unchanged; each decoded value is written once by the kernel and later read once by the consumer. Thus, it breaks the pipeline: decoded data are first materialized in memory, and only then consumed by a separately launched relational kernel.

(Global-stage) Tile decompress to bounded global memory. `CB2T` decodes and consumes tiles in the same kernel but stages decoded values in a bounded, reusable global-memory pool. A grid of W worker thread blocks grid-strides over the input; each worker owns one slot containing T four-byte values for each of the K projected columns. The pool therefore occupies $4WKT$ bytes, independent of dataset cardinality. Each thread block decodes the columns using one reusable shared-memory tile, copies them into its slot, executes the tuple loop, and then reuses the slot. Unlike Pre-decom, Global-stage neither materializes the complete decoded dataset nor launches a separate consumer kernel.

(Shared) Tile decompress to shared memory. `CB2T` cooperatively decodes one tile per projected column into the SM scratchpad and feeds the inner tuple loop directly from there (Section 4.3). After a block-wide barrier, the tuple loop reads the decoded values from shared memory. Unlike Global-stage, Shared neither copies these values to global memory nor reloads them before invoking

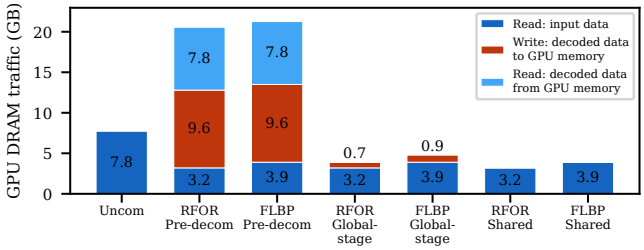


Figure 5: GPU DRAM traffic breakdown for Q1.1 decomposed by phase. Segment colors match the arrows in Figure 4 (blue = HBM read, red = HBM write).

downstream operators. No decoded HBM buffer is ever materialized. The cost is on-chip capacity: shared-memory usage grows with the projected column count, limiting the strategy's reach.

(Register) Decompress into registers. This variant decodes one column at a time into a reusable shared-memory tile and then moves each thread's values into registers, reducing the shared-memory footprint from K tiles to one at the cost of additional register pressure. FLBP supports direct per-position decoding, while RFOR additionally requires cumulative run boundaries and a per-thread search. Register placement benefits simple queries on the A40, but not on the A100, where its register pressure offsets the shared-memory savings, confirming the findings of [21]. We therefore omit it from Figures 4 and 6, but further analyze it in Section 5.5.

5.4 GPU DRAM traffic decomposition

We profile GPU DRAM traffic with NVIDIA Nsight Compute [6] (NCU), collecting DRAM reads and writes across all kernels in the query. We restrict the analysis to Q1.1, which is scan-dominated and join-free: join queries exchange intermediate results through the GPU cache, and NCU's per-kernel serialization evicts this shared state between replays, generating re-fetches that do not occur during normal execution and would render bandwidth attribution meaningless.

Each bar in Figure 5 separates input reads, decoded-value writes, and decoded-value reads. The RFOR and FLBP input segments use their resident compressed sizes, 3.2 and 3.9 GB, respectively; the remaining values represent physical HBM traffic measured by NCU. Uncompressed execution reads 7.76 GB from HBM.

Pre-decom writes the full 9.60 GB decompressed dataset into GPU memory buffers and re-reads it from the query kernel. Including the compressed-input read, this produces 20.60 GB of physical HBM traffic for RFOR and 21.13 GB for FLBP, respectively 2.65× and 2.72× the uncompressed baseline.

For Global-stage, we size the worker pool to expose all available thread-block parallelism while keeping the staging buffer bounded; for Q1.1, it occupies only 63.7 MB. Global-stage stores logically and reloads the complete 9.60 GB decoded stream. Surprisingly, the workers repeatedly access the same portion of the 63.7 MB array using the GPU cache. NCU reports L2 hit rates of 79.1% for RFOR and 74.2% for FLBP, so only 0.70 and 0.90 GB of the logical writes reach HBM. Total physical HBM traffic is 3.98 GB for RFOR and 4.78 GB for FLBP.

Shared removes the global-memory handoff: the tuple loop consumes each decoded tile directly from shared memory before the tile is overwritten. Its bars therefore contain only the compressed input: 3.2 GB for RFOR and 3.9 GB for FLBP.

The main result is that avoiding whole-dataset materialization and fusing decompression and query execution removes most HBM traffic due to decompression. Relative to Pre-decom, Global-stage reduces physical HBM traffic by 5.2× for RFOR and 4.4× for FLBP, while Shared reduces it by 6.3× and 5.6×.

5.5 Comparing encoders and decompression placements against uncompressed Proteus

Figure 6 reports end-to-end query execution time and GPU memory footprint, including the referenced base columns and any placement-specific HBM staging or materialization. For Proteus with `CB2T` we show the fastest configuration `tpb` × `ipt` among: (128×8), (256×4), and (512×2) for FLBP, and (64×8), (128×4) and (256×2) for RFOR.

SSB. Across Q1.1–Q4.1, Shared provides the best compressed runtime in all eight query–codec cases. On scan-intensive Q1.1, FLBP-Shared matches Uncom at 19.7 ms while reducing footprint from 9.6 to 3.8 GB; RFOR-Shared reaches 3.3 GB but takes 36.9 ms because its multi-layer decoder is more expensive. As joins and aggregation add work, decompression is increasingly amortized: on Q2.1–Q4.1, FLBP-Shared ranges from 2% faster to 5% slower than Uncom, while RFOR's overhead falls from 87% on Q1.1 to 6–22%. Wider pipelines nevertheless increase resource pressure: for the selected FLBP-Shared kernels, spill stores/loads rise from 44/56 bytes on Q2.1 to 64/80 on Q3.1 and 112/136 on six-column Q4.1. Global-stage retains a footprint close to Shared but is 33–80% slower. It publishes and reloads every decoded value through the reusable slot in global address space.

TPC-H. Q5 exercises a six-relation pipeline centered on the large SF100 `lineitem–orders` join, carrying fact tuples through five hash probes before aggregation. FLBP-Shared reduces footprint from 11.53 to 5.46 GB and runs in 139 ms versus 110.5 ms for Uncom. RFOR provides similar compression but takes 172 ms, consistent with its lower decoder throughput (284 versus 653 GB/s). Q12 exercises the same large join in a shallower, filter-heavy pipeline. FLBP-Shared obtains a 2.66× footprint reduction at 95.2 ms, only 10% above Uncom; RFOR reaches 2.70× at 121 ms. Global-stage takes 197/205 ms on Q5 and 138/141 ms on Q12 under FLBP/RFOR. Even for these larger and complex join pipelines, eliminating per-column shared staging with Global-stage does not improve runtime.

Stack. Stack's real-world distributions produce the highest compression ratios in the evaluation, ranging from 3.2× to 4.1×. On Q11, FLBP-Shared reduces footprint from 0.597 to 0.168 GB while matching Uncom (8.97 versus 8.99 ms). Wider Q14 scans 21 columns across seven relations; repeated decode before its dictionary-ID and range predicates increases FLBP-Shared to 30.0 ms and RFOR-Shared to 36.6 ms, versus 22.9 ms for Uncom, while retaining 3.19× and 3.33× compression. The bounded Global-stage allocation is proportionally visible at this scale: it adds 88–175 MB on Q11 and 167–334 MB on Q14.

We profile each configuration with NCU, and also report register spill store/load bytes (from `ptxas` at compile time) in Table 3 as diagnostic evidence.

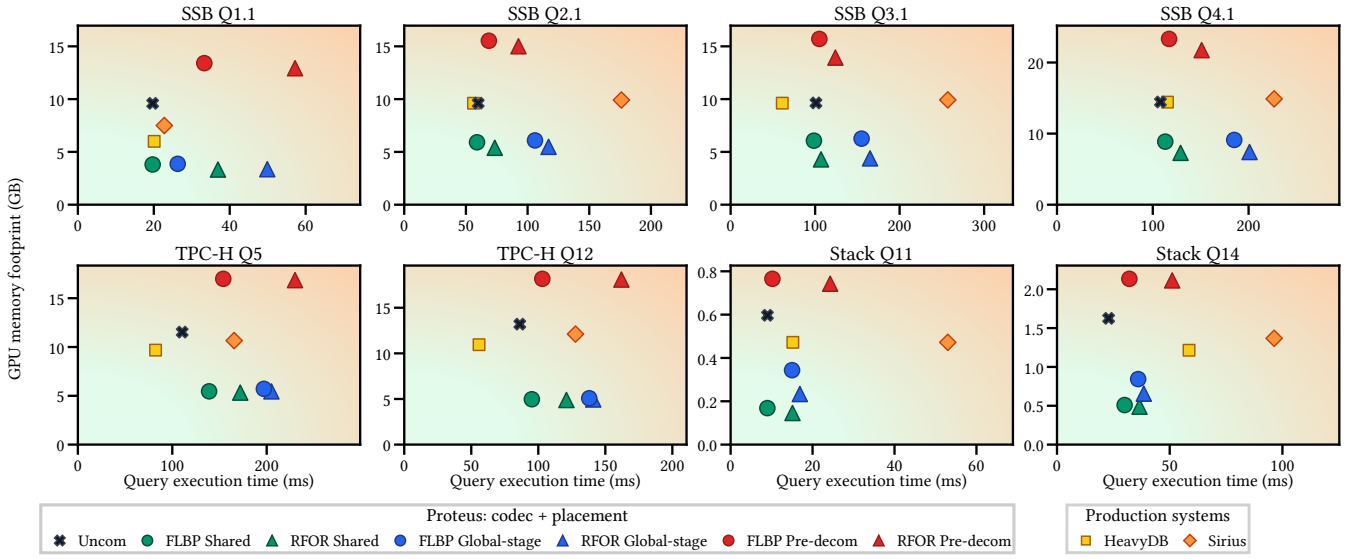


Figure 6: GPU memory footprint vs. query execution time for SSB Q1.1–Q4.1, TPC-H Q5 and Q12, and Stack Q11 and Q14.

Shared vs Registers. SHARED consistently outperforms REGISTER, which matched it on only a few queries and was never faster. GPU A100 profiles show why: occupancy is constrained by register pressure rather than shared-memory capacity, so moving decoded values into registers does not free additional occupancy on this hardware. REGISTER therefore incurs its extra spilling cost with no compensating occupancy gain. Even GLOBAL-STAGE, which reduces the shared-memory footprint by 41–83%, leaves occupancy unchanged. We therefore use SHARED as the default placement.

Launch Configuration. Table 3 sweeps the `tpb`/`ipt` split of the launch configuration - thread-level parallelism versus work per thread [82] - at fixed tile sizes (512 for RFOR and 1024 for FLBP), on the two SSB endpoints: scan-dominated Q1.1 and join-heavy Q4.1. For RFOR, `ipt` = 4 is a single, query-independent optimum: it wins at *every* query in our suite (SSB, TPC-H, Stack). RFOR's decode is register-hungry, and spilling grows monotonically with `ipt`: `ipt` = 8 spills heavily; `ipt` = 4 relieves this pressure by lowering work per thread and raising threads per block, which increases occupancy; `ipt` = 2 gains no further occupancy yet leaves too little work per thread to amortize synchronization. For FLBP, the optimum `ipt` instead varies between 8 and 4 across the same suite. Register pressure explains the SSB queries: Q1.1 spills nothing at any geometry, so more work per thread is effectively free and `ipt` = 8 wins, whereas Q4.1, which spills at every geometry, prefers `ipt` = 4. Unlike RFOR, however, this rule does not generalize beyond SSB: a few queries, e.g., TPC-H Q5 (not shown), spill a high number of registers yet prefer the higher `ipt` = 8 and get slower at the lower-pressure `ipt` = 4, so no single fixed geometry can be recommended the way it can for RFOR. In both codecs, moving past the sweet spot in either direction is never beneficial, consistent with the broader observation that maximizing thread count is not the same as maximizing throughput [69, 82].

Summary. Across the eight queries, SHARED is the fastest compressed placement in all 16 query–codec cases. Averaging per-query percentages, FLBP-SHARED reduces footprint by 54% with only 8% runtime overhead, while RFOR-SHARED reduces it by 60% at a 45% overhead. GLOBAL-STAGE retains 46%/56% footprint reductions for FLBP/RFOR but incurs 62%/88% runtime overhead: L2 absorbs most staged traffic, but has higher latency than SM-local shared memory/L1 and global instructions remain. SHARED avoids both this handoff and PRE-DECOM's full decoded materialization. Our guidance is therefore to decode into shared memory by default. The remaining decision, launch configuration, does not generalize to a single rule: for RFOR, we recommend `tpb` = 128, `ipt` = 4; for FLBP, the optimum varies between 128×8 and 256×4; we therefore recommend restricting tuning to these two configurations.

Table 3: `tpb` ×`ipt` sweep for SHARED: RFOR (512-value tiles) and FLBP (1024) on SSB Q1.1/Q4.1. Spill S/L: register-spill store/load bytes; bold: fastest per query.

		Q1.1			Q4.1		
Codec	Metric	`ipt` = 8	`ipt` = 4	`ipt` = 2	`ipt` = 8	`ipt` = 4	`ipt` = 2
RFOR	`tpb`	64	128	256	64	128	256
	Spill S/L (B)	136/172	44/44	8/8	152/176	112/136	100/124
	Exec. time (ms)	49.6	**36.9**	59.6	275.0	**129.0**	153.0
FLBP	`tpb`	128	256	512	128	256	512
	Spill S/L (B)	0/0	0/0	0/0	156/180	112/136	100/124
	Exec. time (ms)	**19.7**	24.2	34.6	150.0	**119.0**	151

5.6 Production GPU database baseline

In Figure 6 we also compare Proteus against two external GPU database baselines (described in Section 2.3): Sirius (commit `8d8353a`, DuckDB `v1.5.2`) and HeavyDB (commit `da23f70`). The two systems bracket the GPU-engine design space and let us anchor Proteus's runtimes against architectural extremes. For Sirius, we use `gpu_processing`, the in-memory path, which matches our GPU-resident setting. We exclude `gpu_execution`/Super Sirius because

it introduces a different Parquet/task/tiered-memory regime. Both paths use a vectorized, libcudf-based execution model: operators materialize columnar outputs in device memory, whereas compiled engines fuse pipelines so intermediate values flow through registers or shared memory. This trades higher inter-operator memory traffic for avoiding per-query compilation overhead [32, 53].

The reported intervals for all three systems exclude query compilation and SQL planning; they cover GPU pipeline execution through result materialization, making the comparison independent of each system's SQL front-end overhead.

HeavyDB and Sirius are configured to use the narrowest integer types available in their current implementations. This gives the production baselines their strongest coarse-grained footprint reduction. In Figure 6, for example, in SSB Q1.1 `lo_quantity` ranges over $[1, 50]$ and `lo_discount` over $[0, 10]$. HeavyDB stores them as `INT8`, reducing footprint from 9.6 to 6.0 GB, while Sirius lacks `INT8` and uses `INT16`, yielding 7.2 GB. Column-level narrowing helps but remains bounded by one physical type per column. Proteus `CB2T` instead compresses at tile granularity, exploiting local distributions within each column.

Across the eight queries in Figure 6, fixed-width narrowing reduces HeavyDB's and Sirius's base-column footprints by 15% and 8% on average relative to Proteus-Uncom. Nevertheless, FLBP-Shared uses 46% less memory than HeavyDB and 50% less than Sirius, while RFOR-Shared uses 53% and 57% less, respectively. HeavyDB is faster than Proteus-Uncom on four queries and is only 3% slower in geometric mean, whereas Sirius is 2.37× slower. HeavyDB's advantage on TPC-H Q5/Q12 reflects specialized GPU plans that use direct-index perfect-hash joins and keyless aggregations; Proteus uses general build/probe pipelines followed by partial and final aggregation. Sirius's operator-at-a-time execution instead materializes intermediate results between operators, which is particularly costly for the join-heavy queries. Thus, these results compare equivalent queries under each production system's selected physical plan rather than isolating identical join implementations.

Summary. Across all eight queries, HeavyDB remains runtime-competitive through specialized physical plans, while Sirius is slower than Proteus-Uncom on every query; neither matches CB2T's footprint. Averaging per-query percentages, FLBP-Shared incurs only 8% runtime overhead over Proteus-Uncom while using 46% less GPU memory than HeavyDB and 50% less than Sirius. Thus, CB2T combines fused compiled execution with a resident footprint unattainable through fixed-width type narrowing alone.

5.7 Specialized compressed-execution baselines

To isolate CB2T from the general benefit of compressed execution, we compare Proteus with the open-source, tile-based Crystal implementations of GPU-RFOR [79] and FastLanes-GPU [21]. Over SSB (the only available workload for Crystal) the specialized compressed kernels are 2.7-4.7× faster than their Proteus counterparts. Most of this gap exists independently of compression: uncompressed Crystal is already 2.0–4.1× faster because its queries use fixed, hand-written CUDA execution structures and query-specific data structures. Normalizing each compressed configuration to its own uncompressed baseline therefore better isolates decompression cost. RFOR changes Crystal's runtime by +31–94%, compared with +6–87% in Proteus. FLBP changes Crystal's runtime by −47% to +66%, whereas Proteus remains between −2% and +5%. Consequently, `CB2T` 's advantage is not lower absolute kernel time, but achieving the same footprint reduction with lower RFOR overhead and substantially more stable FLBP performance across query shapes. It thus brings the space benefits of specialized compressed execution into a general compiled engine while keeping composability.

6 FUTURE WORK

This work opens three promising directions. First, compiled GPU engines can make decompression placement hardware- and workload-aware: the compiler already observes the projected columns, operator shape, backend, and target SM resources, but selecting just-in-time the best `threads_per_block`, `items_per_thread`, and decode location requires models that capture the interaction between decompression and the fused query pipeline. Second, compression semantics can become compiler-visible. Encoding metadata such as tile minimum value, bit widths, and run boundaries could be represented in the generated LLVM IR and consumed by optimization passes before NVPTX lowering, enabling predicates to skip tiles or avoid decoding values that cannot contribute to the result [91]. Third, as query pipelines increasingly span heterogeneous hardware, the abstraction introduced in this paper can be extended from compressed scans to compressed data exchange. A counterpart operator, TupleToCompressedBlocks (`T2CB`), could consume the standard tuple interface produced by a pipeline on one device and emit compressed blocks, making compression a compiler-inserted boundary around pipeline breakers or device crossings [34]. This would allow intermediate results to be compressed before materialization or transfer to another compute unit, reducing HBM footprint and PCIe/NVLink traffic without manually designing hybrid CPU–GPU data-flow strategies [61]. In such distributed pipelines, the compiler could choose not only where operators run, but also which representation crosses each boundary.

7 CONCLUSION

This paper fills the compiled, tuple-at-a-time-over-compressed design space by making decompression a first-class physical operator in a JIT-compiled GPU engine. `CB2T` confines encoding-specific logic behind reusable backend hooks, so a new compression format is a single backend rather than a pipeline rewrite, and brings decompression within the compiler's reach, opening automatic optimizations that hand-written kernels cannot express. Executing directly over compressed data reduces both memory traffic and footprint, letting queries fit within smaller GPU-memory budgets and more workloads share the same device.

ACKNOWLEDGMENTS

We thank Milos Duric, Charles Benello, and Arturo Paras for their help with preliminary experiments. This work was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) for the Prodasys project, a European Research Council (ERC) Advanced Grant.

REFERENCES

[1] 2021. BlazingSQL. https://github.com/BlazingDB/blazingsql. Accessed: 2026-07-31.

[2] 2022. Oracle Cloud Cost Estimator. https://www.oracle.com/cloud/costestimator.html. Accessed: 2026-07-31.

[3] 2024. HeavyDB. https://github.com/heavyai/heavydb. Accessed: 2026-07-31.

[4] 2024. NVIDIA nvCOMP. https://docs.nvidia.com/cuda/nvcomp/. Accessed: 2026-07-31.

[5] 2025. NVIDIA Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/. Accessed: 2026-07-31.

[6] 2025. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. Accessed: 2026-07-31.

[7] 2026. AWS Pricing Calculator. https://calculator.aws/#/createCalculator/ec2-enhancement. Accessed: 2026-07-31.

[8] 2026. GetDeploying Cloud Cost Comparison. https://getdeploying.com/. Accessed: 2026-07-31.

[9] 2026. Google Cloud Pricing Calculator. https://cloud.google.com/products/calculator. Accessed: 2026-07-31.

[10] 2026. Ibis. https://ibis-project.org/. Accessed: 2026-07-31.

[11] 2026. Lambda Cloud. https://lambda.ai/. Accessed: 2026-07-31.

[12] 2026. LLVM Passes. https://llvm.org/docs/Passes.html. Accessed: 2026-07-31.

[13] 2026. Microsoft Azure Pricing Calculator. https://azure.microsoft.com/en-us/pricing/calculator/. Accessed: 2026-07-31.

[14] 2026. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/. Accessed: 2026-07-31.

[15] 2026. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: 2026-07-31.

[16] 2026. NVIDIA Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/. Accessed: 2026-07-31.

[17] 2026. NVVM IR Specification. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html. Accessed: 2026-07-31.

[18] 2026. RAPIDS cuDF. https://github.com/rapidsai/cudf. Accessed: 2026-07-31.

[19] 2026. TPC-H Homepage. https://www.tpc.org/tpch/. Accessed: 2026-07-31.

[20] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[21] Azim Afroozeh, Lotte Felius, and Peter Boncz. 2024. Accelerating GPU Data Processing using FastLanes Compression. In *Proceedings of the 20th International Workshop on Data Management on New Hardware* (Santiago, Chile) *(DaMoN '24)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3662010.3663450

[22] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. 2012. Chapter 4 - Building an Efficient Hash Table on the GPU. In *GPU Computing Gems Jade Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 39–53. https://doi.org/10.1016/B978-0-12-385963-1.00004-6

[23] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[24] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 283–296. https://doi.org/10.1145/1559845.1559877

[25] Nils Boeschen, Tobias Ziegler, and Carsten Binnig. 2024. GOLAP: A GPU-in-Data-Path Architecture for High-Speed OLAP. *Proc. ACM Manag. Data* 2, 6, Article 237 (Dec. 2024), 26 pages. https://doi.org/10.1145/3698812

[26] Antonio Boffa, Roberto Di Cosmo, Paolo Ferragina, Andrea Guerra, Giovanni Manzini, Giorgio Vinciguerra, and Stefano Zacchiroli. 2025. On the compressibility of large-scale source code datasets. *Journal of Systems and Software* 227 (2025), 112429. https://doi.org/10.1016/j.jss.2025.112429

[27] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. 2022. A Learned Approach to Design Compressed Rank/Select Data Structures. *ACM Trans. Algorithms* 18, 3, Article 24 (Oct. 2022), 28 pages. https://doi.org/10.1145/3524060 Accessed: 2026-07-31.

[28] Martin Boissier. 2021. Robust and budget-constrained encoding configurations for in-memory database systems. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 780–793. https://doi.org/10.14778/3503585.3503588

[29] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proc. VLDB Endow.* 13, 12 (July 2020), 2649–2661. https://doi.org/10.14778/3407790.3407851

[30] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR (Biennial Conference on Innovative Data Systems Research)*. 225–237.

[31] 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. https://doi.org/10.1145/2882903.2882936

[32] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 441–454. https://doi.org/10.14778/3632093.3632107

[33] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with Roaring bitmaps. *Softw. Pract. Exper.* 46, 5 (May 2016), 709–719. https://doi.org/10.1002/spe.2325

[34] 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. https://doi.org/10.14778/3303753.3303760

[35] EPFL DIAS. 2026. Proteus: A Database Engine for Heterogeneous Environments. https://github.com/epfl-dias/proteus. Accessed: 2026-07-31.

[36] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database compression on graphics processors. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 670–680. https://doi.org/10.14778/1920841.1920927

[37] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[38] Henning Funke and Jens Teubner. 2020. Data-parallel query processing on non-uniform data. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 884–897. https://doi.org/10.14778/3380750.3380758

[39] Haralampos Gavriilidis, Lennart Behme, Sokratis Papadopoulos, Stefano Bortoli, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2022. Towards a Modular Data Management System Framework. In *1st International Workshop on Composable Data Management Systems, CDMS@VLDB 2022, Sydney, Australia, September 9, 2022*, Satyanarayana R. Valluri and Mohamed Zaït (Eds.). https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper2_HaralamposGavriilidis.pdf

[40] Mateusz Gienieczko, Maximilian Kuschewski, Thomas Neumann, Viktor Leis, and Jana Giceva. 2025. AnyBlox: A Framework for Self-Decoding Datasets. *Proc. VLDB Endow.* 18, 11 (July 2025), 4017–4031. https://doi.org/10.14778/3749646.3749672

[41] Andrea Guerra, Giorgio Vinciguerra, Antonio Boffa, and Paolo Ferragina. 2025. Learned Compression of Nonlinear Time Series with Random Access . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1579–1592. https://doi.org/10.1109/ICDE65448.2025.00122

[42] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (Dec. 2009), 39 pages. https://doi.org/10.1145/1620585.1620588

[43] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proc. VLDB Endow.* 15, 11 (July 2022), 2811–2825. https://doi.org/10.14778/3551793.3551833

[44] Sven Hepkema, Azim Afroozeh, Charlotte Felius, Peter Boncz, and Stefan Manegold. 2025. G-ALP: Rethinking Light-weight Encodings for GPUs. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3736227.3736242

[45] 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. https://doi.org/10.14778/3705829.3705856

[46] Zezhou Huang, Krystian Sakowski, Hans Lehnert, Wei Cui, Carlo Curino, Matteo Interlandi, Marius Dumitru, and Rathijit Sen. 2025. GPU Acceleration of SQL Analytics on Compressed Data. arXiv:2506.10092 [cs.DB] https://arxiv.org/abs/2506.10092

[47] Yi Jiang, Hamish Nicholson, Viktor Sanca, and Anastasia Ailamaki. 2026. Data Movement-Aware GPU Sharing for Data-Intensive Systems. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. www.cidrdb.org. https://vldb.org/cidrdb/2026/data-movement-aware-gpu-sharing-for-data-intensive-systems.html

[48] Michael Jungmair and Jana Giceva. 2025. Towards Designing Future-Proof Data Processing Systems. *Proc. VLDB Endow.* 18, 11 (July 2025), 3988–3995. https://doi.org/10.14778/3749646.3749669

[49] 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, 3, Article 187 (June 2025), 25 pages. https://doi.org/10.1145/3725324

[50] 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 (Sept. 2025), 4350–4363. https://doi.org/10.14778/3749646.3749698

[51] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. 2012. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware* (Scottsdale, Arizona) *(DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 55–62. https://doi.org/10.1145/2236584.2236592

[52] 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. https://doi.org/10.14778/2994509.2994516

[53] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 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. https://doi.org/10.14778/3275366.3284966

[54] Greg Kimball, Karthikeyan Natarajan, Devavret Makkar, Bradley Dice, and Muhammad Haseeb. 2025. Accelerating Velox with RAPIDS cuDF. VeloxCon 2025. https://prestodb.io/wp-content/uploads/presto-users/Accelerating-Velox-with-RAPIDS-cuDF-VeloxCon-April-2025.pdf Accessed: 2026-07-31.

[55] Laurens Kuiper. 2025. Query Engines: Gatekeepers of the Parquet File Format. https://duckdb.org/2025/01/22/parquet-encodings Accessed: 2026-07-31.

[56] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[57] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data* (Santiago, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 5–17. https://doi.org/10.1145/3626246.3653368

[58] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 311–326. https://doi.org/10.1145/2882903.2882925

[59] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[60] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: balancing I/O and GPU bandwidth in big data analytics. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[61] Yinan Li, Bailu Ding, Ziyun Wei, Lukas M. Maas, Momin Al-Ghosien, Spyros Blanas, Nicolas Bruno, Carlo Curino, Matteo Interlandi, Craig Peeper, Kaushik Rajan, Surajit Chaudhuri, and Johannes Gehrke. 2025. Scaling GPU-Accelerated Databases Beyond GPU Memory Size. *Proc. VLDB Endow.* 18, 11 (July 2025), 4518–4531. https://doi.org/10.14778/3749646.3749710

[62] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proc. ACM Manag. Data* 2, 1, Article 65 (March 2024), 28 pages. https://doi.org/10.1145/3639320

[63] Antonio Mallia, Michał Siedlaczek, Torsten Suel, and Mohamed Zahran. 2019. GPU-Accelerated Decoding of Integer Lists. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 2193–2196. https://doi.org/10.1145/3357384.3358067

[64] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[65] Gonzalo Navarro. 2022. Indexing Highly Repetitive String Collections, Part I: Repetitiveness Measures. *ACM Comput. Surv.* 54, 2 (2022), 29:1–29:31. https://doi.org/10.1145/3434399

[66] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[67] Hamish Nicholson, Konstantinos Chasialis, Antonio Boffa, and Anastasia Ailamaki. 2025. The Effectiveness of Compression for GPU-Accelerated Queries on Out-of-Memory Datasets. In *Proceedings of the 21st International Workshop on Data Management on New Hardware (DaMoN '25)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/3736227.3736240

[68] Hamish Nicholson, Aunn Raza, Periklis Chrysogelos, and Anastasia Ailamaki. 2023. HetCache: Synergising NVMe Storage and GPU acceleration for Memory-Efficient Analytics. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p84-nicholson.pdf

[69] NVIDIA Corporation. 2026. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html. Accessed: 2026-07-31.

[70] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.), Vol. 5895. Springer, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[71] 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. https://doi.org/10.14778/3425879.3425890

[72] Johns Paul, Shengliang Lu, and Bingsheng He. 2021. Database Systems on GPUs. *Found. Trends Databases* 11, 1 (July 2021), 1–108. https://doi.org/10.1561/1900000076

[73] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[74] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (June 2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[75] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[76] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[77] Mark A. Roth and Scott J. Van Horn. 1993. Database compression. *SIGMOD Rec.* 22, 3 (Sept. 1993), 31–39. https://doi.org/10.1145/163090.163096

[78] 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. https://doi.org/10.1145/3318464.3380595

[79] Anil Shanbhag, Bobbi W. Yogatama, Xiangyao Yu, and Samuel Madden. 2022. Tile-Based Lightweight Integer Compression in GPU. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1390–1403. https://doi.org/10.1145/3514221.3526132

[80] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[81] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. https://www.vldb.org/pvldb/vol17/p3694-saxena.pdf

[82] Vasily Volkov. 2010. Better Performance at Lower Occupancy. GPU Technology Conference (GTC). https://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf Accessed: 2026-07-31.

[83] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. 2025. GSST: Parallel string decompression at 191 GB/s on GPU. In *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Rotterdam, Netherlands) *(CHEOPS '25)*. Association for Computing Machinery, New York, NY, USA, 8–14. https://doi.org/10.1145/3719330.3721228

[84] Bowen Wu, Dimitrios Koutsoukos, and Gustavo Alonso. 2025. Efficiently Processing Joins and Grouped Aggregations on GPUs. *Proc. ACM Manag. Data* 3, 1, Article 39 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709689

[85] Youyang Xia, Feng Zhang, Junda Pan, Yihao Liu, Jiawei Guan, Huanchen Zhang, and Xiaoyong Du. 2026. L3: A GPU-Native Co-Designed Data Format for Learned Lossless Lightweight Compression. *Proceedings of the ACM on Management of Data* 4, 3 (May 2026), 1–27. https://doi.org/10.1145/3802078

[86] Bobbi Yogatama, Yifei Yang, Kevin Kristensen, Devesh Sarda, Abigale Kim, Adrian Cockcroft, Yu Teng, Joshua Patterson, Gregory Kimball, Wes McKinney, Weiwei

Gong, and Xiangyao Yu. 2026. Rethinking Analytical Processing in the GPU Era. In *16th Conference on Innovative Data Systems Research, CIDR 2026, Chaminade, CA, USA, January 18-21, 2026*. www.cidrdb.org. https://vldb.org/cidrdb/2026/rethinking-analytical-processing-in-the-gpu-era.html

[87] Xiangyao Yu. 2025. NVIDIA CUDA-X Powers the New Sirius GPU Engine for DuckDB, Setting ClickBench Records. https://developer.nvidia.com/blog/nvidia-gpu-accelerated-sirius-achieves-record-setting-clickbench-record/ Accessed: 2026-07-31.

[88] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. https://doi.org/10.14778/2536206.2536210 Accessed: 2026-07-31.

[89] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. *Proc. VLDB Endow.* 17, 2 (2023), 148–161. https://www.vldb.org/pvldb/vol17/p148-zeng.pdf

[90] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data* 3, 4, Article 245 (Sept. 2025), 27 pages. https://doi.org/10.1145/3749163

[91] Yu Zhang, Feng Zhang, Yani Liu, Huanchen Zhang, Jidong Zhai, Wenchao Zhou, and Xiaoyong Du. 2026. Enabling Tile-Based Direct Query on Adaptively Compressed Data With GPU Acceleration. *IEEE Transactions on Parallel and Distributed Systems* 37, 2 (2026), 410–426. https://doi.org/10.1109/TPDS.2025.3639485

[92] Pinxue Zhao, Hailin Zhang, Fangcheng Fu, Xiaonan Nie, Qibin Liu, Fang Yang, Yuanbo Peng, Dian Jiao, Shuaipeng Li, Jinbao Xue, Yangyu Tao, and Bin Cui. 2025. MEMO: Fine-grained Tensor Management For Ultra-long Context LLM Training. *Proc. ACM Manag. Data* 3, 1, Article 53 (Feb. 2025), 28 pages. https://doi.org/10.1145/3709703