L3: A GPU-Native Co-Designed Data Format for Learned Lossless Lightweight Compression

YOUYANG XIA, Renmin University of China, China
FENG ZHANG*, Renmin University of China, China
JUNDA PAN, Renmin University of China, China
YIHAO LIU, Tsinghua University, China
JIAWEI GUAN, Renmin University of China, China
HUANCHEN ZHANG, Tsinghua University, China
XIAOYONG DU, Renmin University of China, China

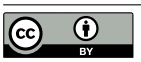
Learned Compression achieves strong CPU performance but lacks a GPU-native format, limiting its use in GPU analytics. We present L3, a GPU-native Learned Lossless Lightweight Compression format that enables end-to-end on-device processing with efficient compression, high-throughput decompression, and fast random access on GPU.On NVIDIA GPUs, a warp is a group of 32 threads; we refer to each thread as a *lane* (lane id 0–31), and call a layout *lane-major* when each lane's words are stored contiguously. L3 introduces three tightly coupled components built around the SLAP Vertical layout. First, the *L3 Storage Layout (SLAP)* stores bit-packed residual streams in a *lane-major* organization, i.e., residual words are laid out lane by lane so each warp lane consumes a contiguous word sequence in memory, exploiting the GPU L1 sector cache for implicit prefetching and high reuse during unpacking. Second, the *Warp-Cooperative Learned Decompression Module* maps each partition to one thread block and decodes warp tiles using per-lane bit readers, branchless bit extraction, and a bit-exact no-FMA FP64 finite-difference predictor. Third, the *GPU-Native Learned Compression Pipeline* builds adaptive partitions via bulk delta-bits analysis, scan/compaction, and an odd-even GPU merge loop, then packs residuals directly into the final SLAP Vertical layout on the device. L3 achieves high performance on modern GPUs. It encodes 3–6× faster than Tile and FastLanes-GPU and sustains 1.08–1.90 TB/s decompression throughput, comparable to the fastest lightweight GPU codecs. On correlated datasets, L3 reaches up to 77× compression while remaining competitive on weakly correlated inputs. For random access, L3 maintains 1.2–2.6 Billion queries/s and outperforms Tile-DFOR/Tile-RFOR by 5–10×. On SSB with unified query plans, L3 achieves the lowest average latency (1.14 ms), matching or outperforming state-of-the-art GPU baselines.

CCS Concepts: • **Information systems** → **Data compression**; • **Computer systems organization** → *Parallel architectures*.

Additional Key Words and Phrases: Lightweight compression, GPU, learned compression, lossless

*Feng Zhang is the corresponding author of this paper.

Authors' Contact Information: Youyang Xia, xiayouyang@ruc.edu.cn, Renmin University of China, Beijing, China; Feng Zhang, fengzhang@ruc.edu.cn, Renmin University of China, Beijing, China; Junda Pan, xm_jarden@ruc.edu.cn, Renmin University of China, Beijing, China; Yihao Liu, liuyihao24@mails.tsinghua.edu.cn, Tsinghua University, Beijing, China; Jiawei Guan, guanjw@ruc.edu.cn, Renmin University of China, Beijing, China; Huanchen Zhang, huanchen@tsinghua.edu.cn, Tsinghua University, Beijing, China; Xiaoyong Du, duyong@ruc.edu.cn, Renmin University of China, Beijing, China.



ACM 2836-6573/2026/6-ART201
https://doi.org/10.1145/3802078

ACM Reference Format:
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. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 201 (June 2026), 27 pages. https://doi.org/10.1145/3802078

1 Introduction

Modern data analytics ecosystems are founded on open columnar storage formats. While formats like Parquet [6] and ORC [5] have become cornerstones for data lakes and data sharing, they were designed over a decade ago for CPU-centric architectures and batch-oriented workloads. Since then, the hardware and workload landscape has evolved toward heterogeneous, accelerator-rich systems and interactive, high-throughput analytics. This evolution exposes an architectural mismatch: legacy formats couple heavyweight, serial compression pipelines and CPU-oriented memory layouts that cannot exploit today's massively parallel hardware. Bridging this gap requires rethinking the format layer itself to enable GPU-efficient ingestion, selective access, and scan throughput.

Learned Compression [36] introduced the *Model + Delta* paradigm, fitting predictive models to value sequences and storing residuals (deltas) for superior ratio and query performance. It generalizes classical predictive encodings such as FOR, RLE, and Delta [1, 20, 33, 73], achieving up to 23× higher compression ratios and 5.2× faster queries than traditional methods. *To realize these gains in modern, GPU-accelerated analytics, the compression stage itself must reside on the device; otherwise PCIe round-trips and serial CPU encoders erase the margin and break end-to-end parallelism.*

GPUs have already proven to be powerful engines for analytical workloads through massive parallelism [57, 59, 64, 69] and are now integral to OLAP engines, stream processors, and key–value stores [17, 23, 24, 38, 63]. Yet most data formats still depend on *CPU-based compression*, creating a severe ingest and transformation bottleneck. While prior work has emphasized *decompression* acceleration, efficient *compression*—particularly learned, lightweight, and lossless (LLL)—is equally crucial. GPU-side LLL is vital for three reasons: (1) it enables high-speed *on-device ingestion and transformation*, avoiding PCIe and CPU serialization; (2) lightweight, parallel compression keeps data resident on the GPU, improving bandwidth utilization and end-to-end throughput; and (3) on-GPU learned modeling adapts to local data characteristics in real time, boosting ratios without sacrificing decode speed. Prior studies echo this trend: Fang et al. [14] showed up to 10× faster loading via GPU compression; Lal et al. [32] improved effective bandwidth with entropy-aware coding; Tian et al. [58] achieved multi-fold Huffman speedups via warp cooperation; and Chen et al. [9] reported over 300× throughput gains—together indicating that GPU-based LLL is an important building block for GPU-accelerated lossless analytics.

However, all existing GPU-native formats cannot integrate learned compression effectively. Giddy [50] lacks fine-grained random access, while Gompresso [55] trades ratio for parallelism. Tile [52] and FastLanes-GPU [4], though fast, perform compression on CPUs and rely on fixed, rule-based models. This disconnection between learned modeling and GPU execution prevents full exploitation of GPU throughput and memory locality.

Developing a GPU-native learned compression format faces three fundamental challenges. (1) Traditional formats treat compression as an offline stage, isolating metadata from query execution and preventing fine-grained predicate pushdown. (2) The resulting compressed layouts from learned models often lack predictable structure for coalesced GPU access and selective decompression. (3) Model fitting and adaptive partitioning in learned compression are inherently serial, conflicting with SIMT execution. Overcoming these challenges requires a unified design that merges compression, layout, and selective query processing requirements within a GPU-parallel framework.

We present **L3**, a GPU-native **l**earned **l**ossless **l**ightweight compression format that combines learned modeling with a GPU-friendly physical layout and high-throughput GPU kernels. L3 introduces three core innovations. First, the *L3 Storage Layout (SLAP)* organizes the bit-packed residual streams in a *SLAP Vertical* (lane-major) layout that exploits sector-based GPU L1 caches for implicit prefetching during unpacking. Second, a *Warp-Cooperative Learned Decompression Module* decodes the Vertical layout using *warp tiles* and branchless bit extraction, sustaining near-peak throughput via SLAP-driven L1 reuse. A warp tile is a fixed group of $T = 32 \cdot V$ values mapped to one warp: lane ℓ reconstructs indices $\{\ell, \ell + 32, \ldots, \ell + (V - 1) \cdot 32\}$ while consuming a contiguous per-lane residual bitstream. Third, a *Delta-Bits Driven Parallel Cost-Optimal Partitioning and GPU-Resident Encoding Pipeline* (Cost-Optimal) builds adaptive partitions via bulk delta-bits analysis, scan/compaction, and batched GPU merge, then packs residuals directly into the final SLAP vertical layout on the device. As part of the format metadata, L3 can optionally store per-partition bounds (e.g., min/max) to enable predicate-based pruning for selective queries.

L3 encodes 3–6× faster than Tile [52] and FastLanes-GPU [3, 4] and sustains 1.08–1.90 TB/s decompression throughput—comparable to Tile/FastLanes-GPU and 2–6× higher than nvCOMP—while preserving high compression efficiency. On strongly correlated data (`Linear`, `Normal`, `Libio`), L3 achieves 15.4×, 13.8×, and 77.2× compression; on less correlated/high-entropy inputs (`Books`, `Facebook`, `Medicare`), it remains competitive (1.68×–3.3×).

In summary, we make the following contributions:

- We identify the gap between CPU-oriented learned compression frameworks (e.g., LeCo [36]) and GPU-native analytics, showing that existing formats lack the architectural design to exploit on-device parallelism and memory locality for compression and query execution.
- We propose L3, a GPU-native learned lossless lightweight format that extends the Model + Delta paradigm to a GPU-optimized architecture in which, once inputs reside on device, the main compression, decompression, and random-access query path executes on the GPU, with only minor host-side control.
- We conduct comprehensive evaluations across synthetic, real-world, and SSB analytical workloads, comparing L3 with state-of-the-art GPU systems and demonstrating consistent gains in compression ratio, decompression throughput, and end-to-end query performance.

2 Background and Motivation

The necessity for a new data format arises from a fundamental architectural mismatch between legacy columnar formats and the modern data analytics paradigm, which is now dominated by GPU computing and advanced compression techniques. This section details these components and exposes their inherent conflicts.

2.1 GPU-Centric Analytics and Its Demands

GPUs were originally designed for graphics rendering and image processing but are now widely adopted in diverse domains, including data management systems [8, 12, 15–17, 24, 37, 38, 49, 63, 71, 72]. With thousands of lightweight cores organized into streaming multiprocessors (SMs), GPUs execute hundreds of threads in parallel under the SIMT model. Fine-grained thread-level parallelism and fast on-chip shared memory enable them to hide global memory latency and sustain high throughput. A single high-end GPU can deliver compute capacity once achievable only by CPU clusters.

Leveraging GPUs for Learned Compression harnesses this massive parallelism for in-situ model fitting and residual encoding, achieving high-throughput compression and decompression while minimizing CPU–GPU data transfers [52]. This integration unites the modeling adaptiveness of

Learned Compression with GPU-level throughput, paving the way for real-time large-scale analytics. However, this performance is predicated on algorithms and data layouts that respect the GPU's architecture—a principle that legacy, CPU-centric formats were not designed to follow.

2.2 Learned Compression and The Conflict with Legacy Formats

Classical schemes such as *Frame-of-Reference (FOR)*, *Run-Length Encoding (RLE)*, and *Delta Encoding* [20, 33, 73] can be interpreted as early forms of Learned Compression. Modern Learned Compression techniques [36] generalize this idea by segmenting data into short partitions, fitting simple predictive models (e.g., linear regressors), and compressing only the prediction errors instead of raw values to achieve significant space savings. As shown in Figure 1, a linear model is fitted to values y_1, y_2, and y_3 to produce predictions $\hat{y}_1, \hat{y}_2$, and $\hat{y}_3$. The residuals $\delta_1 = y_1 - \hat{y}_1, \delta_2 = y_2 - \hat{y}_2$, and $\delta_3 = y_3 - \hat{y}_3$ are then stored and compressed.

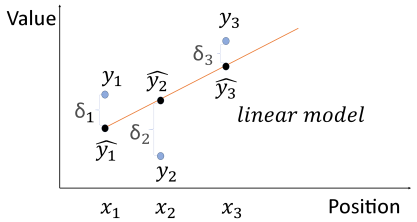


Fig. 1. Learned Compression with linear model

A key strength of this approach is *adaptive partitioning*, which dynamically adjusts segment boundaries to better match local data trends. This is superior to fixed-length partitioning, which can misalign with data characteristics and degrade model fit. To achieve this, state-of-the-art methods like LeCo employ a greedy *variable-length partitioning* algorithm based on split–merge heuristics [36]. As shown in Figure 2, this process makes sequential decisions based on encoding cost until an optimal layout is found. The algorithm operates in two phases. In the *split phase* (Figure 2a), LeCo sequentially grows partitions by fitting a regressor and extrapolating to the next value. When the prediction error exceeds a threshold ("Delta > Threshold"), a split is triggered. Because the algorithm processes data online without lookahead, it cannot foresee that post-jump values may form a new stable trend. This leads to over-segmentation: a region that could be captured by a single model is fragmented into multiple small segments (Figure 2b: Temp Seg A and B). The *merge phase* (Figure 2c) compensates by combining adjacent fragments when a unified model yields lower total cost. Since each partition stores model parameters (θ_0, θ_1) as metadata, merging eliminates redundant overhead while recovering the global trend ("Merge Cost < $\sum$(Temp Segments)").

However, this adaptivity—the core of Learned Compression's effectiveness—creates a direct conflict. The sequential nature of the split-merge algorithm is fundamentally at odds with the parallel execution model of GPUs. Furthermore, this dynamic partitioning logic clashes with the rigid, fixed-block structure of traditional columnar formats, which were designed for simpler, uniform data chunks. This presents a critical trade-off: the parallel efficiency of older schemes versus the superior compression ratio of modern, but sequential, algorithms. A GPU-native learned format must resolve this conflict by combining the high compression efficiency of modern Learned Compression with a design that is fundamentally parallel-friendly for GPUs.

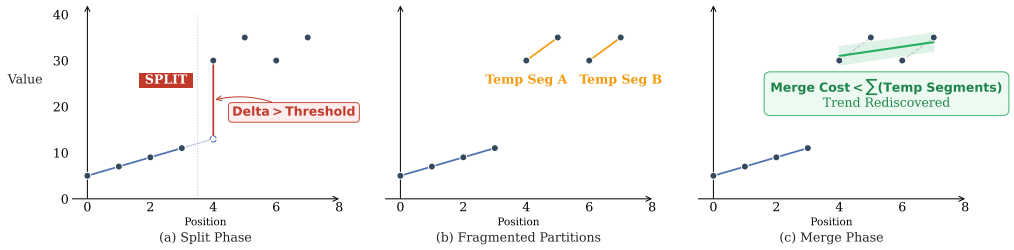


Fig. 2. Illustration of LeCo's variable-length partitioning algorithm.

2.3 Design Challenges

The background illustrates that a new format is necessary, as simply adapting existing algorithms or frameworks is insufficient. Our encoder draws inspiration from LeCo [36], a CPU-oriented learned compression system that introduced the Model+Delta paradigm and adaptive partitioning for exploiting serial correlations in columnar data. However, the underlying assumptions of CPU-centric algorithms conflict with the core principles of high-performance GPU computing. Enabling an efficient, GPU-accelerated learned format required overcoming three major technical challenges in the format's specification itself.

Challenge 1: Polynomial-Aware Learned Decoding Requires Sequential Reconstruction, Conflicting With Coalesced GPU Memory Access. To achieve high compression across datasets with diverse correlation patterns, the format must support adaptive model selection beyond simple linear predictors, including higher-order polynomial models (POLY2/POLY3) and run-length friendly fallbacks (CONSTANT/RLE). Direct polynomial evaluation introduces per-value floating-point multiplications, which are costly on GPUs. To preserve throughput, LINEAR, POLY2 and POLY3 must be implemented via a finite-difference scheme that avoids per-value multiplications. While this does not introduce dependencies in the stored residuals, it *does* constrain the decoder to consume each lane's residual stream sequentially to advance the difference state. Under a naive horizontal layout that assigns each warp lane a contiguous segment of length L, sequential per-lane decoding yields strided global writes spaced by L (and similarly strided reads), preventing both coalesced loads and coalesced stores. The challenge is therefore to design a cache-aware vertical layout that simultaneously (1) keeps each lane's bitstream contiguous for sequential decoding, (2) restores coalesced writes of reconstructed values, and (3) exploits the sector-based GPU L1 cache so that the unavoidable strided first loads are amortized into subsequent L1 hits.

Challenge 2: Learned Decompression's Per-Partition Logic Leads To Warp Divergence And Uncoalesced Memory Access On GPUs. The per-partition decompression logic of Learned Compression [36], while effective on CPUs, leads to a catastrophic performance breakdown when naively ported to a GPU's SIMT architecture. This is not theoretical; our tests of such a kernel, detailed in Table 1, revealed a dismal 3.13% peak occupancy and less than 0.11% DRAM throughput. The breakdown stems from threads within a single warp processing different partitions, which causes severe warp divergence from 'if/else if' branches as each thread applies a different model or bitwidth. This heterogeneity serializes execution—leaving an average of fewer than 5 active threads per warp—while uncoalesced metadata fetches from global memory explain the abysmal DRAM throughput. Furthermore, bit-level delta extraction degrades into slow, multi-instruction software routines like inefficient 'memcpy' operations. The challenge, therefore, is to design a physical bitstream organization that preserves Learned Compression's semantics while being explicitly

structured to enable a warp-cooperative kernel that sustains uniformity, coalesced memory access, and predictable instruction flow.

Table 1. Low-level GPU efficiency metrics for the naive GPU kernels.

Metric (% of peak unless noted)	**Compression**		**Decompression**	
	Min	Max	**Min**	Max
Achieved occupancy	3.12	3.13	3.12	3.13
Compute throughput	0.39	0.39	0.74	0.76
DRAM throughput	0.10	0.11	0.09	0.10
Memory throughput	0.10	0.11	0.14	0.15
SM Busy	11.41	11.50	16.50	16.67
Issue slots Busy	2.77	2.78	4.33	4.34
Avg. active threads / warp	2.47	2.50	4.33	4.47
Eligible warps / scheduler (warp)	0.11	0.11	0.17	0.17
Warp cycles per instruction	9.01	9.02	5.76	5.78

Challenge 3: Split–Merge Dependency Chains In Learned Compression Hinder Parallel Partitioning On GPUs. LeCo's strong compression ratio relies on an adaptive partitioner whose boundary decisions are tightly coupled with per-partition model fitting and residual bit-cost estimation. Once a boundary is split or merged, the training range of the predictor changes, which reshapes the fitted coefficients and the residual distribution, and thus invalidates the cost signals needed by the next decision. This creates a strict step-by-step dependency chain with irregular control flow, directly conflicting with the GPU's requirement for large batches of independent work. A naive GPU approach that evaluates many candidate boundaries in parallel is also impractical, because each candidate requires re-computing model parameters and re-estimating residual bit-costs over overlapping intervals, quickly amplifying work and memory traffic toward quadratic complexity. As shown in Figure 2, even a small boundary shift can change delta bit-widths and encoding costs, and these changes cascade into different downstream merge outcomes. Therefore, the encoder must replace the original CPU-oriented routine with a parallel-first paradigm that extracts GPU-friendly cost signals in bulk and refines partitions through batched merge passes built on scan, compaction, and ordered materialization, producing high-quality adaptive partitions without sequential decision chains.

3 The L3 Format

3.1 Overview

We illustrate the architecture of the L3 format and its GPU-native ecosystem in Figure 3. L3 is a GPU-native Learned Compression format specification designed for both high compression efficiency and high-throughput decompression on modern GPUs. The specification defines the *SLAP Vertical* physical layout and its metadata, and this paper presents the corresponding parallel GPU pipeline to write and read it.

Modules And Core Design Highlights. L3 consists of three core modules built around the SLAP Vertical layout. First, the *L3 SLAP Layout* (Section 3.2) specifies a *SLAP Vertical* (lane-major) organization for the delta bitstream, tailored to the GPU L1 sector cache to maximize reuse during unpacking, addressing Challenge 1. Second, the *Warp-Cooperative Learned Decompression Module* (Section 3.3) maps each partition to one thread block and decodes warp tiles using per-lane bit readers, branchless bit extraction, and bit-exact no-FMA FP64 finite-difference predictors, addressing

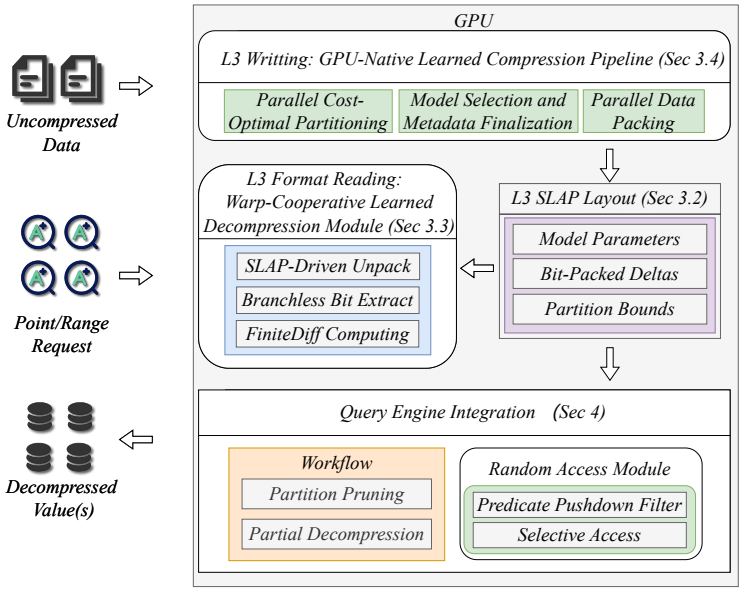


Fig. 3. L3 format and its GPU-native system architecture.

Challenge 2. Third, the *GPU-Native Learned Compression Pipeline* (Section 3.4) implements delta-bits driven parallel cost-optimal partitioning with scan/compaction and odd-even GPU merge (Cost-Optimal), followed by on-device model selection and packing into the final SLAP Vertical layout, addressing Challenge 3.

Query Integration And Compatibility. Beyond full-column decompression, L3 exposes partition boundaries and optional per-partition bounds (e.g., min/max) as part of its metadata, making it straightforward to integrate database-style optimizations such as predicate filtering, partition pruning, and selective access in a GPU query execution engine. These optimizations are orthogonal to the core compression and layout mechanisms, and we use them to demonstrate L3's compatibility with selective query processing.

Workflow. These components enable a two-phase, end-to-end GPU workflow. In the compression phase, raw columnar input is processed by the GPU-native compression engine, producing data in the L3 format directly in device memory. This compressed data can then be stored or transferred to other GPU-resident analytical engines. In the decompression or query phase, data in the L3 format can be fed into the decompression engine for full-column materialization, or consumed by a query execution engine that selectively decodes only the required ranges and optionally uses metadata bounds for pruning.

3.2 L3 SLAP Layout

This section presents the L3 SLAP Format, the core data layout specification for L3's bit-packed delta storage. The L3 format employs the **SLAP (Strided Load Auto-Prefetch)** mechanism to exploit the sector-based L1 cache architecture of modern GPUs, achieving implicit prefetching through carefully orchestrated memory access patterns and significantly reducing memory access latency for decompression operations.

3.2.1 GPU Memory Hierarchy Background. Modern NVIDIA GPUs use a sector-based L1 cache: each 128 B cache line is subdivided into four 32 B sectors, and a warp request loads all sectors that

contain at least one requested address. When the 32 threads access a contiguous 128 B span, the L1 loads the minimum four sectors. In contrast, when the addresses are strided and span a larger region, the L1 loads more sectors even though each thread still requests only a 4 B word.

The SLAP format exploits this behavior by combining a lane-major layout with per-lane sequential decoding. Once a 32 B sector is fetched, up to seven subsequent 32-bit word reads by the same lane reuse it, converting most word loads from DRAM accesses into L1 hits.

3.2.2 L3 SLAP Format Structure. Figure 4 illustrates the L3 SLAP Format for 12-bit packed values. We view the payload as a two-dimensional *lane-by-word* matrix: the *lane* index corresponds to the warp thread (lane i is processed by Thread i, $i \in \{0, \dots, 31\}$), and the *word* index enumerates the consecutive 32-bit words that store the packed residual stream for that lane. A *lane* is simply the per-thread access stream: in figures, lanes may be drawn horizontally or vertically; whichever direction the thread traverses corresponds to the lane. Each lane contains W 32-bit words, where $W = \lceil V \times b/32 \rceil$, V is the number of values per lane, and b is the residual bit-width.

In L3, each lane's packed residual stream starts at a 32-bit word boundary (word-aligned). We enforce this by choosing the tile size V (values per lane) as a multiple of 32 (we use V=64 in our implementation), making $V \cdot b$ divisible by 32 for any integer b; thus $W = \lceil V \cdot b/32 \rceil$ is exact and introduces no per-lane padding.

The key design principle is that **all words belonging to the same lane are stored contiguously in memory**. In other words, each thread's data forms a contiguous block in memory. Formally, for a warp tile processed by a warp of 32 threads, the memory address of word w in lane i is given by:

$$\text{addr}(i, w) = \text{base} + i \times W + w \tag{1}$$

where $w \in \{0, 1, \dots, W-1\}$ indexes the words (columns) and $i \in \{0, 1, \dots, 31\}$ indexes the lanes (rows). This formulation makes explicit that addresses within the same lane (fixed i, varying w) differ by 1 word and are therefore contiguous.

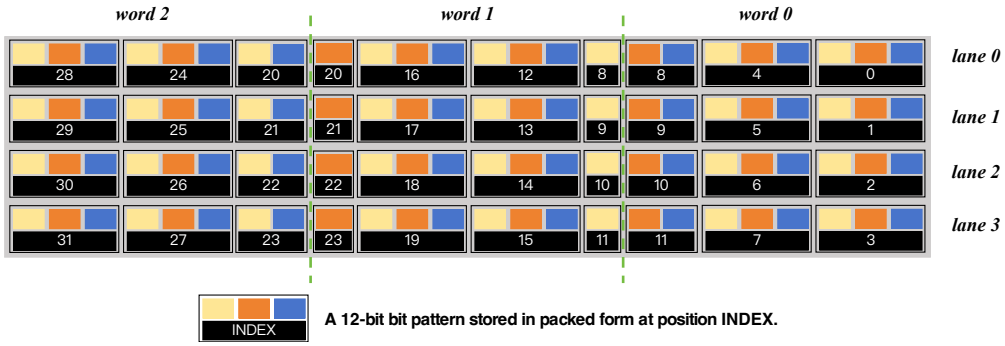


Fig. 4. L3 SLAP layout for 12-bit residuals (lane-major). Rows are lanes; words within a lane are contiguous.

3.2.3 SLAP: Strided Load Auto-Prefetch Mechanism. The performance advantage of the L3 SLAP Format arises from the interaction between its lane-major memory layout and GPU's sector-based L1 cache loading behavior. During decompression, each thread iterates through all words of its lane (horizontally across its row): Thread i reads word 0, then word 1, up to word $(W-1)$, all from Lane i's contiguous memory region. We term the resulting cache behavior the SLAP (Strided Load Auto-Prefetch) mechanism. Figure 5 illustrates this mechanism.

The SLAP mechanism works as follows:

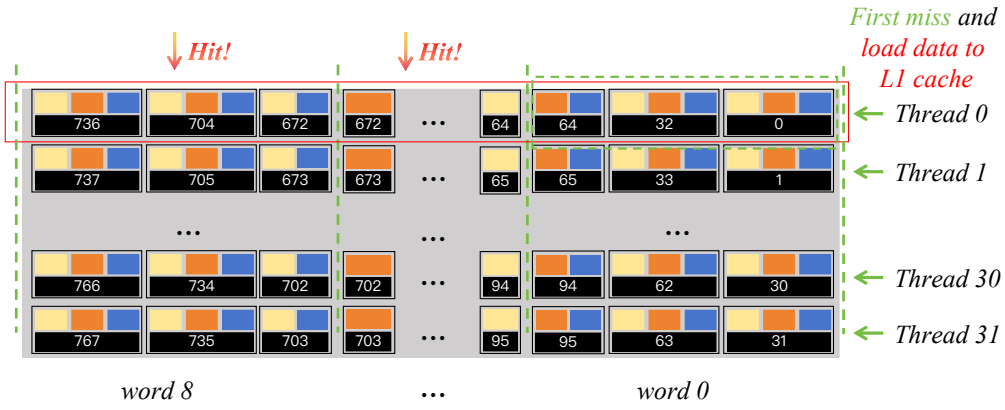


Fig. 5. SLAP mechanism: strided accesses fetch L1 sectors; subsequent within-sector word accesses hit in L1.

First Iteration—Word 0 (Strided Load): When all threads simultaneously access word 0 of their respective lanes (the rightmost column in Figure 5), the memory addresses form a strided pattern: Thread 0 accesses base + 0, Thread 1 accesses base + W, Thread 2 accesses base + $2W$, and so on. The stride between consecutive threads is W words ($W \times 4$ bytes). This warp-wide strided access touches many L1 sectors and fetches the initial sector(s) needed by the per-lane streams.

Subsequent Iterations—Word 1 To Word $(W-1)$ **(Sector Reuse):** In subsequent iterations, each thread advances sequentially within its lane. Since a 32 B sector contains $S = 32\,\mathrm{B}/4\,\mathrm{B} = 8$ consecutive 32-bit words, once the first word of a sector is fetched, the next up to $S-1$ word loads by the same lane typically hit in L1. When a lane crosses a sector boundary (every S words), the first access to the next sector incurs a compulsory miss and brings that sector into L1, which is then reused by the remaining word loads in the same sector.

The key insight is that SLAP trades *per-iteration coalescing* for *per-lane temporal locality*: strided warp accesses fetch the relevant sectors, and the lane-major layout ensures each fetched sector is reused by multiple subsequent word loads.

Synergy with Per-Lane Sequential Bitstream Decoding. SLAP's L1 reuse is reinforced by the Vertical decoder's access pattern: within each warp tile, each thread consumes a contiguous per-lane residual bitstream in order using a buffered bit reader. This per-lane sequential consumption turns the lane-major layout (Equation 1) into a predictable sequence of adjacent word reads, so fetched sectors are reused by the remaining unpack steps.

3.2.4 Theoretical Analysis. The L1 cache hit rate for the L3 SLAP Format can be derived from the sector size. Let $W = \lceil V \times b/32 \rceil$ denote the number of 32-bit words each lane must load, where V is values per lane and b is the bit-width. Let $S = 32\,\mathrm{B}/4\,\mathrm{B} = 8$ denote the number of 32-bit words per L1 sector. Since each lane reads words sequentially, its W words span $\lceil W/S \rceil$ sectors; the first access to each sector is a compulsory miss, and the remaining word accesses within that sector hit in L1, yielding:

$$\text{L1 Hit Rate} \approx 1 - \frac{\lceil W/S \rceil}{W} \tag{2}$$

When $W \leq S$, this reduces to $(W-1)/W$; as W grows, it approaches $1-1/S$. We validate this sector-reuse model experimentally in Section 5.

3.2.5 Latency Advantage. The performance benefit of the L3 SLAP Format stems from the substantial latency difference between L1 cache hits and DRAM accesses. On modern GPUs, L1 cache latency is approximately 28 cycles while DRAM latency exceeds 400 cycles. For W words per lane, SLAP incurs approximately $\lceil W/S \rceil$ DRAM sector fetches and $W - \lceil W/S \rceil$ L1 hits. In contrast, a word-interleaved (FastLanes-style) layout has negligible cross-iteration L1 reuse and thus incurs roughly W DRAM fetches. The resulting latency reduction is:

$$\text{Speedup} \approx \frac{W \times T_{\text{DRAM}}}{\lceil W/S \rceil \times T_{\text{DRAM}} + (W - \lceil W/S \rceil) \times T_{\text{L1}}} \tag{3}$$

Substituting typical latency values ($T_{\text{DRAM}} = 400$ cycles, $T_{\text{L1}} = 28$ cycles), the L3 SLAP Format achieves theoretical speedups ranging from 1.87× at $W = 2$ to 4.44× at $W = 6$. This latency reduction is the fundamental source of L3's performance advantage for bit-unpacking operations.

3.3 L3 Format Reading: Warp-Cooperative Learned Decompression

To efficiently read L3's *SLAP Vertical* physical layout and overcome Challenge 2, we implement a *warp-cooperative, model-aware* GPU decompression kernel (Algorithm 1). Given a partition id *pid*, its metadata record, and the delta bitstream, one thread block materializes $out[s..e)$ by decoding *warp tiles* of $T = 32 \cdot V$ values: each lane sequentially consumes a contiguous per-lane residual stream and writes V outputs at coalesced positions within the tile. Reconstruction follows $v_i = \texttt{base} + u_i$ for `MODEL_FOR_BITPACK` and $v_i = \text{round}(f_\theta(i-s)) + \delta_i$ for `MODEL_LINEAR/POLY2/POLY3`; `MODEL_CONSTANT` uses a dedicated constant/RLE expansion path.

Design. The decoder assigns one thread block per partition and treats a warp tile ($T = 32 \cdot V$ values) as the basic unit of work. Each warp lane consumes a contiguous per-lane residual bitstream in order (required by finite-difference predictors) and writes V reconstructed values to coalesced positions within the warp tile. Uniform (mt, bw) within a partition keeps warps largely lockstep, while the lane-major SLAP layout turns the unavoidable strided first loads into L1 sector reuse for subsequent word reads.

Detailed Algorithm. Algorithm 1 consists of six blocks: (a) *Metadata* (Lines 2–3) loads $[s, e)$ and (mt, bw, off, num_tiles) from a compact per-partition record and lazy-loads the needed model parameters θ; (b) *Constant/RLE* (Lines 4–10) either fills $out[s..e)$ for single-run partitions or decodes the RLE payload and expands runs cooperatively; (c) *Warp-tile loop* (Lines 11–21) decodes modeled partitions tile by tile, where each lane initializes its finite-difference state, computes a lane-local bit offset, and sequentially consumes its per-lane residual stream (with a bw=0 fast path); (d) *Bit extraction* (Line 17) uses `read4` for $bw \leq 32$ and branchless `shift+mask` over a 128-bit window for $bw > 32$; (e) *Predictor update* (Line 19) advances the no-FMA FP64 finite-difference state to match the encoder's evaluation order; (f) *Boundary handling* (Line 22) decodes and writes the remaining values, if any.

Correctness And Complexity Analysis. Residual extraction uses sequential bit addressing: for `delta_bits` ≤ 32, the decoder reads from a 64-bit window (two 32-bit words); for `delta_bits` > 32, from a 128-bit window (two 64-bit words). Both cases apply `shift+mask` followed by sign-extension for signed residuals. Model-based correctness relies on identical FP64 evaluation order via shared no-FMA finite-difference primitives; FOR partitions recover the base losslessly (including `uint64` bit-patterns) and add unsigned deltas. The tail region follows the encoder's bit-offset contract, and padding ensures unconditional window loads remain in-bounds. Complexity is $O(N)$ with $O(1)$ operations per element; metadata loading is $O(1)$ per partition via a compact record. Memory traffic scales with compressed and output sizes; output writes are coalesced, input reads are lane-local and sequential (with optional shared-memory buffering for RLE/tail), enabling high-throughput GPU decompression.

Algorithm 1 Warp-Cooperative Decompression on SLAP Vertical Layout

function DecompressPartition(*pid*, *meta*, *bits*, *out*)
  Load $[s, e)$, mt, bw, off, num_tiles from a compact metadata record
  Lazy-load model params θ required by mt
  if mt = `MODEL_CONSTANT` **then**
    if `num_runs` = 1 **then**
      Fill $out[s..e)$ with constant value; **return**
    end if
    Decode RLE header+residual payload from *bits* into shared memory
    Parallel decode runs → `run_values[]`, `run_counts[]`
    Warp-parallel prefix sum → `run_offsets[]`
    Warp-cooperative expand runs to $out[s..e)$; **return**
  end if
  for each warp tile t assigned to this warp **do**
    Init FP64 finite-diff predictor for this lane (if mt is modeled)
    if $bw = 0$ **then**
      Materialize predictions for this warp tile; **continue**
    end if
    $W \leftarrow \lceil V \cdot bw/32 \rceil$; $bit \leftarrow ((off + t \cdot 32 \cdot W + lane \cdot W) \ll 5)$
    Init lane bit reader at bit
    for $k = 0$ **to** $V - 1$ **do**
      Read packed delta (`read4` for $bw \leq 32$; `shift+mask` for $bw > 32$)
      Reconstruct $out[s + t \cdot T + k \cdot 32 + lane]$
      Advance predictor state (no-FMA FP64 finite-diff accumulation)
    end for
  end for
  Decode remainder values (if any) and write to *out*
end function

3.4 L3 Format Writing: GPU-Native Learned Compression Pipeline

We generate the L3 *SLAP Vertical* format with a GPU-resident learned lossless compression pipeline that keeps intermediate state and the final compressed buffers on device. Concretely, the encoder performs: (i) delta-bits driven cost-optimal partitioning with an odd-even GPU merge loop (Cost-Optimal), (ii) per-partition model fitting and cost-based selection (FOR/LINEAR/POLY2/POLY3, plus unified CONSTANT/RLE), and (iii) prefix-scan layout finalization and warp-tile packing into the final bitstream. Aside from the initial host-to-device input copy and minor host-side control/allocation (e.g., breakpoint materialization and output sizing), this design minimizes CPU–GPU roundtrips.

3.4.1 Delta-Bits Driven Parallel Cost-Optimal Partitioning (Cost-Optimal). L3 uses a **delta-bits driven** cost-optimal partitioner (Algorithm 2) that takes an input column x and produces an ordered partition list with per-partition $(start, end, mt, \theta, bw)$. It is parallel-first: analysis blocks and candidate partitions are processed in bulk on the GPU, and refinement is performed by a batched odd-even merge loop driven by end-to-end storage cost. BestDeltaBits(i)s polynomial-aware (POLY2/POLY3); to avoid FP64 hazards, `uint64` values beyond 2^{53} are guarded by forcing `FOR`.

Cost-Optimal uses a *single threshold* τ (default: 8 bits) to detect breakpoints where residual bitwidth changes sharply, then relies on *cost-driven merging*: neighboring partitions merge only

if doing so reduces total storage cost (compressed bytes plus metadata overhead). This design eliminates manual per-dataset tuning—the cost function automatically adapts to diverse data distributions.

Design. The partitioner follows a parallel-first pipeline: (i) estimate delta-bits per analysis block in parallel, (ii) detect breakpoints and emit an initial partition list via scan/scatter, (iii) fit/select a model per partition, and (iv) iteratively merge neighboring partitions in odd-even phases, using scan/compaction to update the list until convergence.

Algorithm 2 Delta-Bits Parallel Cost-Optimal Partitioning with GPU Merge (Cost-Optimal)

1: **function** PARTITIONCOSTOPTIMAL(x, cfg)
2: **parallel for** analysis block b: $bits[b] \leftarrow$ BESTDELTABITS(x, b, cfg)
3: **parallel for** b: $bp[b] \leftarrow [|bits[b] - bits[b{-}1]| \geq \texttt{cfg}.\tau]$
4: Materialize breakpoint positions BP
5: EMITINITPARTS(BP, cfg.$p_{\min}$) ▷ scan + write $(start, end)$
6: **parallel for** part p: FITSELECT(x, p, cfg)$\rightarrow (mt[p], \theta[p], bw[p])$
7: **for** $r = 1$ **to** cfg.R **do**
8: **parallel for** neighbor pairs: compute merge benefit
9: decide merges (even phase, then odd phase)
10: COMPACT(merge_flags) ▷ scan + scatter
11: **if** no merges **then break**
12: **end if**
13: **end for**
14: ORDERFIX(parts) ▷ GPU sort/gather by start, fix boundaries
15: **return** $parts = \{start, end, mt, \theta, bw\}$
16: **end function**

Detailed Algorithm. Algorithm 2 consists of six blocks: (a) *Delta-bits analysis* (Line 2) computes a cheap, model-aware residual bitwidth estimate $bits[b]$ per analysis block (used only for breakpoint detection); (b) *Breakpoint detection* (Lines 3–4) flags sharp changes $|bits[b]-bits[b-1]| \geq \tau$ and materializes breakpoint positions; (c) *Initial emission* (Line 5) emits an initial $(start, end)$ list via scan/scatter at granularity $p_{\min}$; (d) *Local modeling* (Line 6) fits/selects (mt, θ, bw) per partition by minimizing end-to-end storage cost (metadata + payload, including the RLE payload estimate for `MODEL_CONSTANT`); (e) *Cost-driven refinement* (Lines 7–12) iteratively merges adjacent partitions in odd-even phases and updates the list via scan-based compaction, terminating when no merges remain; and (f) *Ordering* (Lines 13–14) sorts/fixes boundaries and returns an ordered, non-overlapping cover of the input.

Correctness And Complexity Analysis. The pipeline maintains the invariant that partitions are represented as $(start, end)$ pairs that cover disjoint input ranges. EMITINITPARTS(e)mits a contiguous cover within each breakpoint segment; the odd-even merge loop only merges adjacent partitions and uses compaction to produce a conflict-free next list. ORDERFIX(e)nforces a sorted, non-overlapping final cover and repairs boundaries if necessary. Model selection is local to each partition and does not affect the partition coverage invariant. Let M be the number of analysis blocks and P be the number of partitions. Delta-bits estimation and breakpoint detection cost $O(M)$ work; initial emission, per-partition fitting/selection, and each merge round cost $O(P)$ work. With at most R merge rounds, total work is $O(N + M + (R + 1)P)$ and memory is $O(M + P)$; all stages are GPU-parallel via bulk kernels plus scan/compaction.

Partition Size Bounds and Work Balance. The partitioner enforces `min_partition_size` (256 elements) as the initial partition granularity before merging, and `max_partition_size` (8192

elements) as the upper bound after merging. This bounds worst-case imbalance to 32× during decompression (where each partition maps to one thread block). In practice, cost-optimal merging produces only 2–4× variation. We found that explicit load balancing mechanisms degrade performance due to synchronization overhead; the GPU's hardware scheduler naturally balances work across SMs.

3.4.2 Model Selection and Layout Metadata Finalization. Given the partition list and model selections, the encoder computes each partition's payload size (including the optional RLE payload for `MODEL_CONSTANT`) and assigns non-overlapping output regions via prefix scans (Algorithm 3). For `MODEL_CONSTANT`, the single-run case emits no payload; otherwise, it emits a compact cascaded RLE payload (6-word header + two residual streams) that predicts run values and counts with float `LINEAR` models plus bit-packed residuals, and it may additionally emit optional bounds and a consolidated metadata record.

Design. For each partition, the encoder derives (i) the number of full warp tiles and (ii) the total number of 32-bit words needed for its payload. A global prefix scan over these word counts produces `offsets` (word offsets). The encoder also computes a prefix sum over `num_tiles` to map a global warp-tile id to its partition via binary search in the packing kernel.

Algorithm 3 Layout Metadata Finalization

1: **function** FinalizeLayout(*parts*, `cfg`)
2: **parallel for** partition p:
3: $n \leftarrow end[p] - start[p]$
4: **if** $mt[p]$ = `MODEL_CONSTANT` **then**
5: $runs \leftarrow \lfloor \theta[p][0] \rfloor$
6: $word\_cnt[p] \leftarrow$ RLEWords$(runs, \theta[p])$ ▷ 0 if $runs$=1
7: $ntile[p] \leftarrow 0$
8: **else**
9: $ntile[p] \leftarrow \lfloor n/T \rfloor$ ▷ $T = 32 \cdot V$
10: $word\_cnt[p] \leftarrow \lceil (n \cdot bw[p])/32 \rceil$
11: **end if**
12: $off \leftarrow$ `exscan`$(word\_cnt)$; $tile\_pref \leftarrow$ `exscan`$(ntile)$
13: allocate `deltas` (padding, zero-init)
14: **return** $meta = \{off, ntile, tile\_pref\}$ ▷ optional: consolidated metadata + bounds
15: **end function**

Detailed Algorithm. Algorithm 3 consists of six blocks: (a) *Per-partition sizing* (Lines 2–3) derives the element count n; (b) *Constant/RLE sizing* (Lines 4–7) computes the payload word count via RLEWords(a)nd emits zero payload for the single-run case; (c) *Modeled sizing* (Lines 8–10) computes *ntile* and *word_cnt* from (n, bw); (d) *Offset assignment* (Line 12) performs two exclusive scans to produce off and $tile\_pref$ (mapping global tile ids to partitions); (e) *Output allocation* (Line 13) allocates and zero-initializes the word array with padding; and (f) *Metadata output* (Line 14) returns the offsets needed by packing/decoding (and optionally bounds and a consolidated metadata record).

Correctness And Complexity Analysis. The two exclusive scans in Line 12 assign each partition a unique, non-overlapping output region: for any partitions $p \neq q$, the intervals $[off[p],\ off[p] + word\_cnt[p])$ and $[off[q],\ off[q] + word\_cnt[q])$ are disjoint, and their concatenation covers exactly the allocated word array. The scan over *ntile* yields a monotonically increasing $tile\_pref$, so FindPart(c)an map any global tile id to its owning partition. Complexity is $O(P)$ work and

memory: per-partition sizing is $O(1)$, scans are $O(P)$, and allocation is proportional to the final compressed size.

3.4.3 Parallel Packing (Warp Tiles). The packing kernel writes the selected residuals into the SLAP Vertical bitstream entirely on the GPU (Algorithm 4). It maps one block to one warp tile and one thread to one lane; each lane emits a contiguous per-lane residual stream into its partition's assigned output region. Boundary tiles and the `MODEL_CONSTANT` bookkeeping are handled by small auxiliary kernels and omitted; predictors use shared no-FMA FP64 finite-difference accumulation to ensure bit-exact consistency with decoding.

Design. Packing follows the same warp-tile mapping as decoding. Each block packs one warp tile; each thread (lane) emits a contiguous per-lane bitstream of V residuals into its partition's preassigned word range. `atomicOr` resolves cross-word boundaries without serializing the common aligned case.

Algorithm 4 Parallel Packing (Warp Tiles, Vertical)

1: **function** PackTiles(x, $parts$, $meta$, out)
2: $t \leftarrow$ blockIdx.x;$lane \leftarrow$ threadIdx.x ▷ 1 block per tile, 1 thread per lane
3: $p \leftarrow$ FindPart($t, meta.tile\_pref$)
4: $t\_in \leftarrow t - meta.tile\_pref[p]$
5: $bw \leftarrow parts.bw[p]$
6: load (mt, θ)
7: **if** $mt =$ `MODEL_CONSTANT` **then**
8: **return**
9: **end if**
10: $W \leftarrow \lceil V \cdot bw/32 \rceil$
11: $base\_bit \leftarrow ((meta.off[p] + t\_in \cdot 32 \cdot W + lane \cdot W) \ll 5)$
12: **for** $k = 0$ **to** $V - 1$ **do**
13: $idx \leftarrow parts.start[p] + t\_in \cdot T + k \cdot 32 + lane$
14: $\delta \leftarrow$ Residual($x[idx], mt, \theta, idx$) ▷ no-FMA FP64 predictor
15: BitOr(out, $base\_bit + k \cdot bw$, δ, bw) ▷ `atomicOr`
16: **end for**
17: **end function**

Detailed Algorithm. Algorithm 4 consists of six blocks: (a) *Tile/lane mapping* (Lines 2–3) assigns one block to a global warp tile t, maps it to its owning partition p using $tile\_pref$, and derives the in-partition tile id t_in; (b) *Partition parameters* (Lines 4–5) load (mt, θ, bw) and skip `MODEL_CONSTANT` (handled separately); (c) *Bit addressing* (Line 6) computes each lane's base bit offset in the word array from $off[p]$; (d) *Packing loop* (Lines 7–8) enumerates the V lane values and computes each element index; (e) *Residual generation* (Line 9) evaluates the no-FMA finite-difference predictor and forms the signed residual; and (f) *Bit insertion* (Line 10) writes residual bits via `atomicOr` to handle cross-word boundaries.

Correctness And Complexity Analysis. Prefix scans over per-partition word counts assign each partition a unique, non- overlapping region in the final word array; zero-initialization plus `atomicOr` yields correct bitwise composition when writes span word boundaries. Bit-exactness is ensured by using the same no-FMA FP64 finite-difference predictor in both encoder and decoder. The overall work is linear in the number of elements, with additional overhead proportional to the number of analysis blocks and partitions, and all major steps are parallelized on the GPU.

4 Implementation

We implement L3 as a C++/CUDA library whose format layer specifies the SLAP Vertical format and metadata, and whose execution layer provides GPU kernels for format I/O and query primitives (e.g., point/range random access, predicate filtering via partition bounds, partition pruning, and partial decompression). The encoder runs delta-bits driven cost-optimal partitioning, per-partition model selection, and packing on device (Algorithms 2–4); the decoder materializes partitions via warp-cooperative, model-aware unpacking with bit-exact no-FMA FP64 finite-difference predictors (Section 3.3). Once inputs reside on device, both encoding and decoding keep intermediate and final buffers in GPU memory, with only minor host-side control and buffer sizing.

5 Evaluation

In this section, we present a comprehensive empirical evaluation of the L3 format. Our experiments are designed to systematically validate the key design decisions of L3 and demonstrate its end-to-end performance benefits over state-of-the-art CPU and GPU solutions.

5.1 Experimental Setup

Evaluation Methods. We compare L3 against state-of-the-art GPU-native compression frameworks to establish its position in the performance landscape, including the *Tile-based framework* [52] (FOR/DFOR/RFOR), *FastLanes-GPU* [4], and NVIDIA's production-grade *nvCOMP* library [45].

In the context of the *SSB* evaluation, we further include three representative GPU methods that have been widely adopted in prior GPU compression studies [14, 52]: (1) Planner [14], (2) GPU-BP [40], (3) OmniSci(HeavyDB) [23].

Platform. We evaluate L3 on a GPU server equipped with an Intel Core i9-14900X CPU (24 cores, 32 threads), 32 GB of RAM, and an Nvidia H20 GPU (Hopper architecture, 96 GB of memory, 60 MB L2 cache). The operating system is Ubuntu 22.04.03.

Scalability. L3 supports slicing large columns into GPU-sized chunks; since compression operates on small partitions (256–8192 elements), slicing preserves local characteristics with negligible overhead.

Datasets. We evaluate L3 on all 20 integer datasets from LeCo [36]. Table 2 summarizes the sources and sizes. All of the real world datasets are drawn from SOSD [30], OpenStreetMap [47], Libraries.io [28], MovieLens [22], Kaggle House [27], Public BI [18], and mlcourse.ai [42]. To connect dataset characteristics with model choices, we group datasets by correlation (measured by the R^2 coefficient of a least-squares linear fit) and repetition. *Group A (High Correlation)* contains sorted sequences with $R^2 > 0.9$ (e.g., `Linear` has $R^2 = 1.0$ and `Normal` has $R^2 \approx 0.95$); real SOSD indices (`Books`, `Wiki`) and `Libio` also fall in this regime. It also includes datasets better captured by low-order polynomials (e.g., `Planet` reaches $R^2 \approx 0.96$ under quadratic fitting). *Group B (Medium Correlation)* covers moderately regular data ($0.5 < R^2 < 0.9$), such as irregularly sampled timestamps (`ML`) and stochastic event streams (`Poisson`). *Group C (Low Correlation / High Entropy)* stresses robustness where linear models offer limited benefit ($R^2 < 0.5$): `Facebook` has near-zero correlation, while `OSM` and `Medicare` are large, irregular ID-like columns; L3 falls back to FOR-equivalent behavior while retaining GPU-friendly decoding. *Group D (Non-Linear Patterns)* includes synthetic piecewise curves where quadratic fitting improves the linear-fit R^2 by > 0.1 (`Polylog`, `Exp`, `Poly`), motivating polynomial models. *Group E (High Repetition / RLE-Friendly)* contains discrete attributes with 94–99% identical adjacent values (`Adult`, `Site`, `Weight`), where CONSTANT/RLE dominates over polynomial fitting.

Table 2. Dataset Characteristics. All 20 integer datasets from prior work [36] are included. "Real" denotes data derived from production systems; "Synth" denotes generated data.

Dataset	Source	Items	Type	Real/Synth
Group A: High Correlation (Linear/Near-Linear Trends)				
Linear	Generated	200M	uint32	Synth
Normal	Generated	200M	uint32	Synth
Books	SOSD	200M	uint32	Real
Wikipedia	SOSD	200M	uint64	Real
Planet	OpenStreetMap	200M	uint64	Real
Libio	Libraries.io	200M	uint64	Real
Group B: Medium Correlation				
ML	UCI	14M	uint64	Real
Poisson	Generated	87M	uint64	Synth
Group C: Low Correlation (High Entropy)				
Facebook	SOSD	200M	uint64	Real
OSM	SOSD	800M	uint64	Real
MovieID	MovieLens	20M	uint32	Real
House	Kaggle	2.2M	uint64	Real
Medicare	Public BI	1.5B	uint64	Real
Group D: Non-Linear Patterns				
Cosmos	Generated	1M	int32	Synth
Polylog	Generated	10M	uint64	Synth
Exp	Generated	200M	uint64	Synth
Poly	Generated	200M	uint64	Synth
Group E: High Repetition / RLE-Friendly				
Adult	mlcourse.ai	30K	uint32	Real
Site	mlcourse.ai	250K	uint32	Real
Weight	mlcourse.ai	25K	uint32	Real

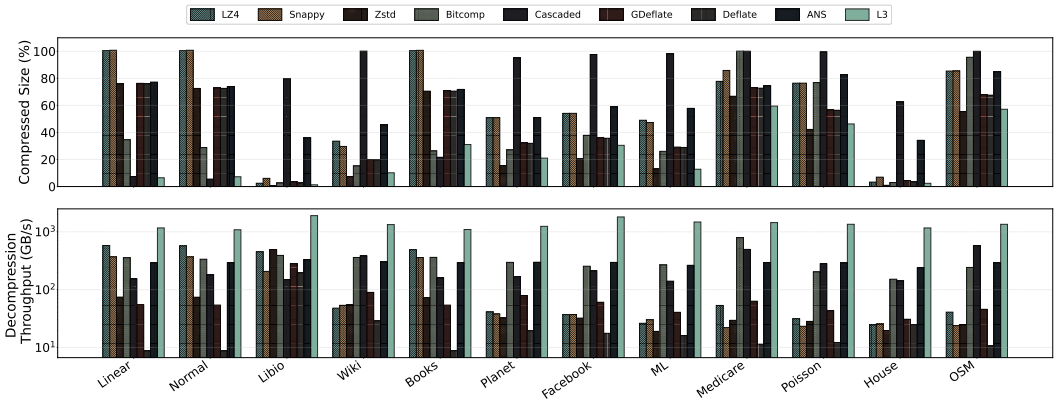


Fig. 6. L3 vs nvCOMP on 12 datasets (H20). **Top:** compressed size (%). **Bottom:** decompression throughput.

5.2 End-to-End Performance Analysis

This section evaluates the end-to-end performance of the L3 format, focusing on its core design trade-offs: decompression speed, space efficiency, compression (writing) speed, and random access latency. For readability, Figures 6, 7, and 8 visualize 12 representative datasets (Linear, Normal, Libio, Wiki, Books, Planet, Facebook, ML, Medicare, Poisson, House, OSM). We report the remaining 8 datasets (Movieid, Cosmos, Polylog, Exp, Poly, Adult, Site, Weight) in Figure 10 and Table 3, and summarize all 20 datasets in Table 4.

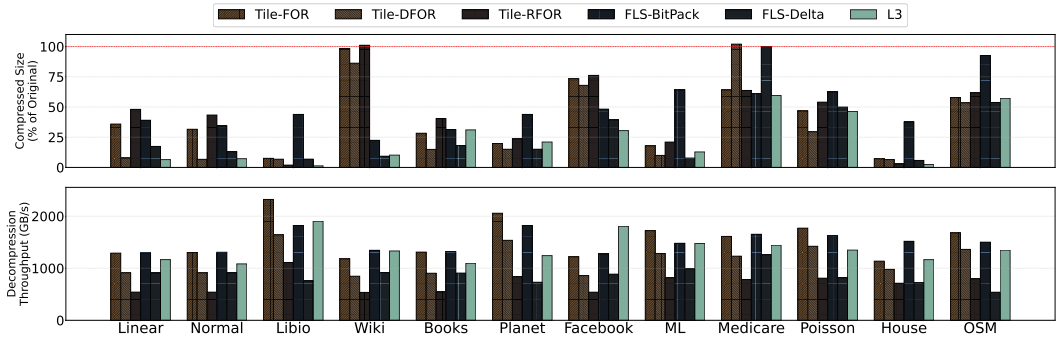


Fig. 7. L3 vs Tile-GPU and FastLanes-GPU on 12 datasets (H20). **Top:** compressed size (%). **Bottom:** decompression throughput.

5.2.1 Decompression Performance and Space Efficiency. A primary design goal of L3 is to resolve the trade-off between the high compression ratios of modern Learned Compression and the high throughput of simpler GPU codecs. Figure 6 and Figure 7 report decompression throughput and compression ratios.

SLAP Validation. To validate the cache-reuse model in Section 3.2, we profile the decoder using NVIDIA Nsight Compute. On H20, the measured L1 sector hit rates match Equation 2 within 3% error (e.g., 47–83% for $W = 2$–6 in our microbenchmarks), and we observe the same sector-based loading behavior on RTX 4090 and RTX PRO 6000.

Throughput. L3 sustains 1.08–1.90 TB/s across all 12 datasets, comparable to the fastest lightweight integer codecs on GPUs such as Tile-FOR (1.14–2.32 TB/s) and FastLanes-BitPack (1.28–1.82 TB/s). It averages 1.37 TB/s, peaking at 1.90 TB/s on *Libio* and bottoming out at 1.08 TB/s on *Normal*. Despite employing a more sophisticated learned model, L3 stays within the top throughput tier due to the integrated design between its storage layout and the Warp-Cooperative Decompression engine. Figure 7 further shows that L3 is close to Tile-FOR/FastLanes-BitPack and is consistently faster than multi-layer variants (Tile-DFOR/RFOR and FastLanes-Delta), which drop to 0.54–1.64 TB/s. In contrast, nvCOMP's general-purpose codecs—including LZ4, Snappy, Zstd, and ANS—reach only 8–795 GB/s (Figure 6); e.g., on *Linear* L3 achieves 1.17 TB/s whereas LZ4 reaches 579 GB/s, and on *Wiki* L3 exceeds 1.33 TB/s while Cascaded delivers 383 GB/s. This confirms that L3's warp-cooperative decoding achieves full memory-bandwidth utilization and scales efficiently across data patterns.

Compression Ratio. L3 not only sustains high throughput but also maintains superior compression efficiency. On datasets with strong serial correlations, such as *Linear*, *Normal*, and *Libio*, L3 achieves compression ratios of 15.4×, 13.8×, and 77.2×, respectively—competitive with Tile, FastLanes-GPU and nvCOMP's best ratios of 13.7× (Cascaded), 17.9× (Cascaded), and 150.1× (Zstd). On less correlated datasets (*Facebook*, *Medicare*), L3 remains competitive, achieving ratios between

1.68× and 3.3× and staying within the range of Tile/FastLanes-GPU and nvCOMP variants. These results demonstrate that learned modeling enables L3 to exploit fine-grained statistical redundancy that generic codecs overlook, achieving both compact representation and high decode speed.

Summary. Overall, L3 attains 2–6× higher decompression throughput than nvCOMP and provides a competitive ratio–throughput frontier versus Tile and FastLanes-GPU. It achieves a balanced operating point between throughput and ratio, approaching the speed of the simplest GPU codecs while providing the compression efficiency of advanced learned models, validating L3's integrated design philosophy for GPU data processing.

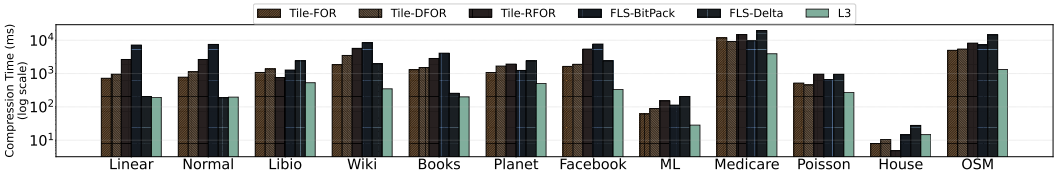


Fig. 8. Encoding time: Tile-GPU, FastLanes-GPU, and L3 on 12 datasets (H20, log scale).

5.2.2 Compression (Writing) Performance. A GPU-native format is only practical if it can be generated efficiently on the GPU. Figure 8 and Figure 9 compare the encoding time of L3's on-GPU write engine against other frameworks. Against Tile and FastLanes-GPU, whose compression is CPU-based, L3's GPU-native pipeline is 3–6× faster on average, finishing in 15–3939 ms compared to their longer execution times. This demonstrates the benefit of an end-to-end, GPU-resident workflow. While general-purpose codecs in nvCOMP (especially Bitcomp and ANS) can have higher raw encode throughput due to their simpler logic, L3 intentionally invests more GPU work into delta-bits driven cost-optimal partitioning (Cost-Optimal) and per-partition model selection. This extra modeling and partition refinement cost pays off in higher compression ratios and enables the query-time optimizations discussed later.

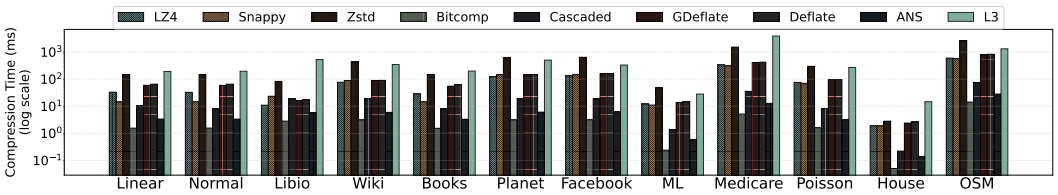


Fig. 9. Encoding time: nvCOMP vs L3 on 12 datasets (H20, log scale).

5.2.3 The Case For Non-Linear Datasets. To understand when L3's learned approach provides the most benefit, we compare compression effectiveness across datasets with varying distribution characteristics (Figure 10). For datasets where simple delta encoders are competitive, such as *Movieid* and *Cosmos*, specialized delta encoders (Tile-DFOR, FLS-Delta) achieve competitive or better compression ratios. However, L3 consistently delivers the highest decompression throughput on *Movieid* (998.6 GB/s vs. 845.7 GB/s for Tile-DFOR), demonstrating that its learned models can still optimize for speed even when compression gains are modest.

The advantage of L3 becomes pronounced on non-linear datasets. On *Polylog*, L3 achieves only 3.62% of original size—nearly 2× better than Tile-DFOR (6.04%) and FLS-Delta (6.69%)—because its

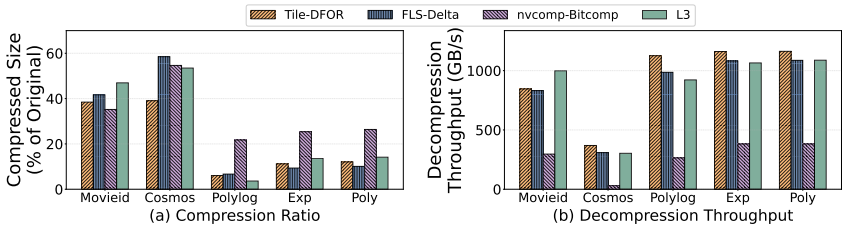


Fig. 10. Representative datasets: compressed size (%) vs decompression throughput (GB/s).

polynomial model families can capture the underlying mathematical structure rather than relying on generic delta encoding.

For the highly compressible *Site*, *Weight*, and *Adult* datasets (Group E, Table 3), which contain sorted categorical attributes (e.g., site IDs, body weights, ages) exhibiting over 94% repeated adjacent values, L3 achieves compression ratios comparable to nvcomp-Bitcomp (0.84–4.91% vs. 0.73–3.89%) while delivering 14–15× higher decompression throughput. This demonstrates L3's key design goal: achieving near-optimal compression through learned models while maintaining GPU-friendly decompression performance.

Table 3. Additional datasets: compressed size (% of original) and decompression throughput (GB/s).

Dataset	Tile-DFOR	FLS-Delta	nvcomp	L3
Site	7.04% / 133.2	10.92% / 259.8	3.89% / 11.8	4.91% / 179.7
Weight	5.14% / 16.3	10.52% / 27.4	0.88% / 1.4	0.95% / 20.1
Adult	5.13% / 19.2	9.81% / 31.7	0.73% / 1.8	0.84% / 25.8

5.2.4 Model Selection Analysis. Table 4 presents the model selection distribution across 20 diverse datasets, revealing how L3's adaptive partitioner tailors compression strategies to underlying data characteristics.

L3 selects models by local regularity: `LINEAR` for near-linear trends, `POLY2`/`POLY3` for smooth nonlinear segments, `CONSTANT`/RLE for long runs, and `FOR` with minimum partitions for high entropy. *Linear* is 100% `LINEAR` ($\Delta b = 2.0$, Avg=4076, 15.4×); *OSM* is 100% `FOR` (Avg=256, $\Delta b = 36.6$, 1.75×); *Medicare* is 100% `FOR` (Avg=256, $\Delta b = 38.1$, 1.68×).

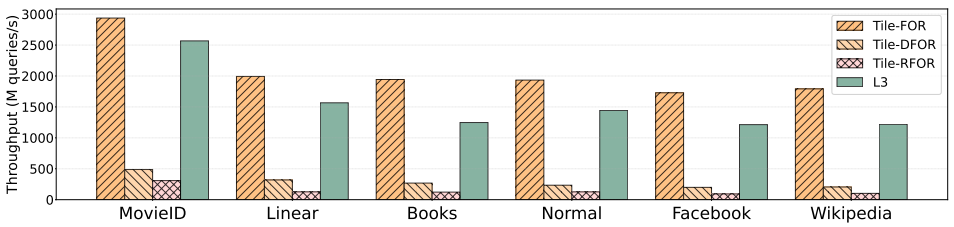


Fig. 11. Random access throughput comparison.

Table 4. Partition statistics and model distribution. Avg=average partition size, Δb=residual bits/element. C/L/P2/P3/F are model distributions. Percentages may not sum to 100 due to rounding.

Dataset	N	Parts	Avg	Δb	C	L	P2	P3	F
Group A: High Correlation									
Linear	200M	49K	4076	2.0	–	100%	–	–	–
Normal	200M	82K	2451	2.0	–	98%	2%	1%	–
Books	200M	411K	487	8.4	–	48%	26%	23%	3%
Wiki	200M	321K	623	5.5	10%	28%	37%	25%	–
Planet	200M	532K	376	11.0	–	42%	20%	15%	23%
Libio	200M	51K	3920	12.7	100%	–	–	–	–
Group B: Medium Correlation									
ML	14M	24K	590	6.1	–	72%	11%	17%	–
Poisson	87M	326K	268	28.6	–	9%	2%	–	89%
Group C: Low Correlation									
Facebook	200M	689K	290	15.9	–	14%	8%	9%	70%
OSM	800M	3.1M	256	36.6	–	–	–	–	100%
Movieid	20M	63K	315	14.1	–	9%	9%	5%	77%
House	2.2M	896	2483	10.1	93%	3%	2%	2%	–
Medicare	1.5B	5.9M	256	38.1	–	–	–	–	100%
Group D: Non-Linear Patterns									
Cosmos	1M	3.9K	256	15.4	–	–	31%	69%	–
Polylog	10M	5.8K	1733	5.7	17%	59%	19%	5%	–
Exp	200M	378K	529	8.5	1%	49%	29%	22%	–
Poly	200M	368K	543	9.8	3%	49%	29%	19%	–
Group E: High Repetition / RLE-Friendly									
Site	250K	120	2083	11.2	88%	3%	7%	3%	–
Weight	25K	7	3572	12.9	100%	–	–	–	–
Adult	30K	9	3333	12.9	89%	–	–	–	11%

5.3 End-to-End System Evaluation on SSB

5.3.1 Parallel Random Access. The L3 format's fine-grained, partitioned layout is designed to support efficient random access, a weakness of traditional coarse-grained formats. Figure 11 compares the random-access (RA) query throughput between L3 and Tile. We focus on Tile as it is a state-of-the-art GPU framework that explicitly supports random access. L3 maintains consistently high throughput (1.2–2.6 B queries/s), comparable to the lightweight Tile-FOR and significantly outperforming the more complex Tile-DFOR and Tile-RFOR variants by 5–10×. These results demonstrate that L3's adaptive, cost-optimal learned partitioning preserves high random-access performance while offering much higher compression efficiency.

To evaluate the end-to-end impact of L3's design choices in a realistic analytical workload, we use the Star Schema Benchmark (SSB) [46]. Following prior GPU database studies [2, 4, 13, 52], we run SSB at a scale factor of 20.

5.3.2 Experimental Setup and Methodology. **Baseline Implementation.** We compare L3 against six compression frameworks. For each baseline, we use the original source code and implement identical SSB query kernels to ensure a fair comparison: (i) *HeavyDB* [23]: a production GPU database; we use its built-in SQL interface. (ii) *Tile* [52]: we use the authors' open-source implementation with FOR/DFOR/RFOR encodings. (iii) *FastLanes-GPU* [4]: we use the official GPU implementation with vectorized bit-unpacking. (iv) *GPU-BP* [40]: we use the original bit-packing library and implement SSB queries following the Crystal [13] operator pattern. (v) *nvCOMP* [45]: we use NVIDIA's cascaded compression API (v3.0.6) with BitComp and ANS layers. (vi) *Planner* [14]: a

cascaded compressor (FOR+Delta+RLE) with cost-based layer selection; since we could not find a public codebase, we re-implemented it following the paper's description.

Query Plan Unification. For selective OLAP queries with typical selectivity rates of 1–15% [29, 48], avoiding unnecessary column decoding is crucial. To isolate the impact of compression format from query execution strategy, all GPU-based implementations (except HeavyDB) use a unified query plan with two key optimizations: (i) *Early Exit*: after each filter stage, we check via `__ballot_sync()` whether any thread in the warp has valid tuples; if not, the warp skips subsequent column decoding entirely. (ii) *Late Materialization*: aggregation columns (e.g., `lo_revenue`, `lo_extendedprice`) are decoded only for rows that pass all filter predicates, avoiding unnecessary decompression work. This unified plan ensures that performance differences stem from the underlying compression/decompression efficiency, not from query execution logic.

Data Preprocessing. SSB contains both numeric and string attributes. For clarity, we refer to integer-typed SSB columns (e.g., `lo_orderkey`, `lo_custkey`, `lo_quantity`) as *integer-heavy* attributes. Following standard practice in GPU databases [13, 52], we apply dictionary encoding to all string columns (e.g., `s_region`, `c_nation`, `p_brand`) during data loading, converting them to integer codes. Date columns (e.g., `lo_orderdate`) are stored as integers in YYYYMMDD format. All frameworks compress and query the same dictionary-encoded integer columns, ensuring identical input data across experiments.

Time Measurement. We measure GPU kernel execution time with data already resident in GPU memory in compressed form. Specifically: (i) For *fused* frameworks (L3, FastLanes-GPU, Tile, GPU-BP, Planner), we report the time for the query kernel that performs decompression and query processing together. All implementations use the same query plan with early-exit and late-materialization optimizations. (ii) For *nvCOMP*, we report the sum of decompression time plus query kernel time, as its API requires fully decompressing columns to global memory before query processing can begin. (iii) For HeavyDB, we report end-to-end query execution time from its profiler. Unless otherwise stated, our cross-framework comparisons (Figure 12) exclude host-to-device transfer and compression time, focusing on the GPU-side execution time under a unified query plan. We additionally report an end-to-end time breakdown (H2D, hash-table build, kernel, total) for L3 and FastLanes-GPU in Table 5. Each query is executed 5 times after a warmup run, and we report the average.

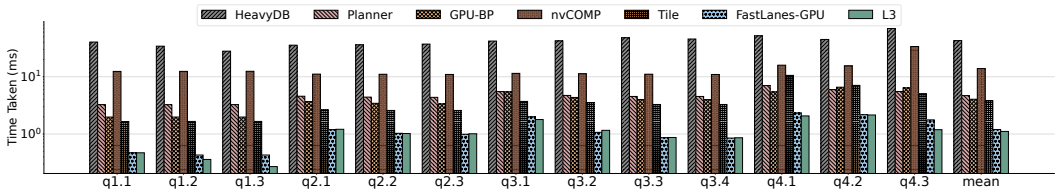


Fig. 12. Execution time per SSB query.

5.3.3 SSB Query Execution. **At A Glance.** Figure 12 reports the GPU-side execution time per query across 13 SSB queries for six baselines—HeavyDB (42.7 ms), *nvCOMP* (13.9 ms), *Planner* (4.7 ms), *GPU-BP* (4.1 ms), and *Tile* (3.8 ms, *FastLanes-GPU* (1.20 ms)—alongside *L3* (1.14 ms). L3 achieves the lowest average latency, outperforming Tile by 3.3× on average and up to 6.1× on Q1.3. Both L3 and FastLanes-GPU complete all queries in under 2.5 ms, setting a new performance ceiling for GPU-native analytical compression.

Table 5. SSB end-to-end time breakdown by query group (ms): H2D transfer, hash-table build (HT), GPU kernel (Kern), and total. L3 vs FastLanes-GPU with unified query plans.

	L3				FastLanes-GPU			
Query	**H2D**	**HT**	**Kern**	**Total**	**H2D**	**HT**	**Kern**	**Total**
Q1.X	32.10	0.00	0.37	32.47	33.70	0.00	0.44	34.14
Q2.X	58.50	0.83	1.24	60.57	61.40	0.81	1.07	63.30
Q3.X	54.30	0.80	1.17	56.30	57.00	0.83	1.20	59.03
Q4.X	84.30	1.42	1.80	87.58	88.50	1.44	2.08	92.06
Avg	**57.30**	**0.79**	**1.14**	**59.23**	**60.15**	**0.79**	**1.20**	**62.13**

L3 Time Breakdown. Table 5 reports an end-to-end breakdown across query groups for both L3 and FastLanes-GPU. Since all baselines use identical query plans (Early Exit + Late Materialization), the key differentiator in the GPU-only comparisons is the *Kernel* time, which reflects the efficiency of fused decompression and query execution. L3's kernel time averages 1.14 ms across all 13 queries, with Q1.X (filter-only) completing in under 0.4 ms. In the end-to-end breakdown, H2D transfer dominates the total time and scales with compressed data size; L3's slightly higher compression ratio yields ∼5% lower H2D time.

Summary. Across the full SSB workload, L3 achieves 37× lower latency than HeavyDB and 3–12× lower than other GPU compression frameworks. The unified query plan ensures these gains stem from L3's format efficiency—learned residuals, adaptive partitioning, and metadata-driven pruning—rather than execution-level differences.

5.4 Format Trade-off Analysis

To clarify the design space of GPU lightweight compression, we provide a comprehensive comparison of feature support and practical trade-offs across formats. Table 6 compares feature support. Only L3 provides all six capabilities: GPU encoding, GPU decoding, random access, adaptive models, adaptive partitioning, and cascaded compression.

Table 6. Comparison of GPU Lightweight Compression Formats. ✓= supported, ✗= not supported.

	nvCOMP	**Tile-GPU**	**FastLanes**	**L3**
GPU Encoding	✓	✗	✗	✓
GPU Decoding	✓	✓	✓	✓
Random Access	✗	✓	✓	✓
Adaptive Models	✗	✗	✗	✓
Adaptive Partition	✗	✗	✗	✓
Cascaded Compression	✓	✓	✗	✓

6 Related Work

From Lightweight To Learned Compression On CPUs. Traditional compression spans dictionary-based [7, 70], entropy-based [53], and general-purpose block compressors (zstd [41], Gzip [19], LZ4 [39]). In analytical databases, lightweight codecs such as PForDelta [73] and FSST [7] enable scan-time decompression and in-place query processing [1, 10, 26] via operator pushdown [11, 21, 34]. Recent work further explores compressed data direct computing, where computation is performed directly on compressed representations [65–68]. Learned Compression advances this

line by fitting regression-style models (also used in learned indexes such as RMI [31] and PGM-Index [16]) and storing residuals, achieving higher ratios [25, 61, 62]. Yet its core mechanism—adaptive, variable-length partitioning [43, 56, 60, 65]—typically relies on serial, CPU-centric algorithms, which does not map cleanly to massively parallel architectures.

Parallel Data Compression On GPU. GPUs have limited memory but high bandwidth, making compression attractive for accelerating analytics. Early GPU systems use multi-pass, cascaded decompression [14, 35], but intermediate writes to global memory can become bottlenecks [52]. Modern designs shift to tile-based/vectorized execution that fuses decompression with query processing [51], enabling specialized bit-packing [40, 52] and data-parallel layouts that remove serial dependencies in Delta and RLE [2].

GPU Databases And Compression-Aware Query Engines. Recent GPU database systems show that compression directly impacts query performance and data movement. HippogriffDB [35] demonstrates GPU-resident OLAP by streaming compressed columns from NVMe into device memory and overlapping I/O, decompression, and kernel execution. Crystal [13], PumpUp [38], and AresDB [54] unify caching, I/O, and columnar GPU execution for real-time analytics, while production engines such as HeavyDB [23] embed vectorized decompression within scan operators. Quantitative studies further highlight this coupling: lightweight delta or bit-pack schemes best utilize PCIe and NVLink bandwidth [44], whereas heavier codecs can underuse SMs. Meanwhile, warp-cooperative entropy coding can still reach multi-terabyte throughput [58]. Collectively, these works establish a principle central to L3: compression should be an in situ GPU stage, not a preprocessing step.

7 Conclusion

We presented L3, a GPU-native format for Learned Lightweight Lossless Compression that rethinks CPU-oriented designs for massively parallel architectures. L3 provides a GPU-resident workflow that, once inputs reside on device, unifies data layout, compression, and query execution within a single framework, with only minor host-side control. By integrating learned modeling with warp-cooperative execution, it removes sequential dependency chains and enables high-throughput, selective decompression on GPUs. Across diverse datasets and workloads, L3 encodes 3–6× faster than Tile and FastLanes-GPU, sustains 1.08–1.90 TB/s decompression throughput, and reaches up to 77× compression on correlated data while remaining competitive on high-entropy inputs. On SSB with unified query plans, L3 achieves the lowest average latency and improves over other GPU compression frameworks by 3–12×. L3 bridges the gap between learned compression and GPU parallelism, establishing a unified, high-efficiency foundation for next-generation analytical systems. Beyond compression, L3's partitioned format metadata (including optional per-partition bounds) and warp-cooperative primitives provide a reusable substrate for in-situ analytics over compressed data.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. U25B2018, 62322213, 62461146205), BAAI, Beijing Science and Technology Project (Z251100008125032), and the State Key Laboratory of Processor, ICT, CAS (No. CLQ202414). Youyang Xia, Feng Zhang, Junda Pan, Jiawei Guan, and Xiaoyong Du are with Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper.

References

[1] 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*. ACM, 671–682. doi:10.1145/1142473.1142548

[2] Azim Afroozeh and Peter Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers Per Second with Scalar Code. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2132–2144. doi:10.14778/3598581.3598587

[3] Azim Afroozeh and Peter Boncz. 2025. The FastLanes File Format. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4629–4643. doi:10.14778/3749646.3749718

[4] 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 (DaMoN '24)* (Santiago, Chile). ACM, 1–11. doi:10.1145/3662010.3663450

[5] Apache Software Foundation. 2013. Apache ORC. https://orc.apache.org/

[6] Apache Software Foundation. 2022. Apache Parquet. https://parquet.apache.org/

[7] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2649–2661. doi:10.14778/3407790.3407851

[8] Cheng Chen, Wenlong Ma, Congli Gao, Wenliang Zhang, Kai Zeng, Tao Ye, Yueguo Chen, and Xiaoyong Du. 2025. GaussDB-AISQL: A Composable Cloud-Native SQL System with AI Capabilities. *Frontiers of Computer Science* 19, 9 (2025), 199608. doi:10.1007/s11704-024-40624-2

[9] Xinyu Chen, Jiannan Tian, Ian Beaver, Cynthia Freeman, Yan Yan, Jianguo Wang, and Dingwen Tao. 2024. FCBench: Cross-Domain Benchmarking of Lossless Compression for Floating-Point Data. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1418–1431. doi:10.14778/3648160.3648180

[10] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *CoRR* abs/2004.09350 (2020). doi:10.48550/arXiv.2004.09350

[11] Dinesh Das, Jiaqi Yan, Mohamed Zait, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee. 2015. Query Optimization in Oracle 12c Database In-Memory. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1770–1781. doi:10.14778/2824032.2824074

[12] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 969–984. doi:10.1145/3318464.3389711

[13] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2432–2444. doi:10.14778/3476249.3476292

[14] Wenbin Fang, Bingsheng He, and Qiong Luo. 2010. Database Compression on Graphics Processors. *Proceedings of the VLDB Endowment* 3, 1–2 (2010), 670–680. doi:10.14778/1920841.1920927

[15] Xiaoyue Feng, Dashan Wei, Chaopeng Guo, and Jie Song. 2025. L2SM: A Query-Optimized Linked LSM-Tree for HTAP Workloads. *Frontiers of Computer Science* 19, 7 (2025), 197606. doi:10.1007/s11704-024-40553-0

[16] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175. doi:10.14778/3389133.3389135

[17] 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*. ACM, 1603–1618. doi:10.1145/3183713.3183734

[18] Bogdan Ghita, Stefan Manegold, and Peter Boncz. 2019. Public BI Benchmark. https://ir.cwi.nl/pub/33082/

[19] GNU Project. 2022. GNU Gzip. https://www.gnu.org/software/gzip/

[20] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the 14th International Conference on Data Engineering*. IEEE, 370–379. doi:10.1109/ICDE.1998.655800

[21] Goetz Graefe and Leonard D. Shapiro. 1990. *Data Compression and Database Performance*. Technical Report. University of Colorado, Boulder, Department of Computer Science. Technical report.

[22] GroupLens Research. 2015. MovieLens 20M Dataset. https://grouplens.org/datasets/movielens/20m/

[23] HEAVY.AI. 2022. HeavyDB (formerly OmniSciDB). https://www.heavy.ai/product/heavydb

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

[25] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Cetintemel. 2020. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 1733–1746. doi:10.1145/3318464.3389734

[26] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, 843–856. doi:10.1145/3448016.3457283

[27] Kaggle. 2022. USA Real Estate Dataset. https://www.kaggle.com/datasets/ahmedshahriarsakib/usa-real-estate-dataset

[28] Jeremy Katz. 2020. Libraries.io Open Source Repository and Dependency Metadata. doi:10.5281/zenodo.3626071

[29] Rini T. Kaushik. 2014. FlashQueryFile: Flash-Optimized Layout and Algorithms for Interactive Ad Hoc SQL on Big Data. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kaushik

[30] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2019. SOSD: A Benchmark for Learned Indexes. *CoRR* abs/1911.13014 (2019). doi:10.48550/arXiv.1911.13014

[31] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 489–504. doi:10.1145/3183713.3196909

[32] Sohan Lal, Jan Lucas, and Ben Juurlink. 2017. E^2MC: Entropy Encoding Based Memory Compression for GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1119–1128. doi:10.1109/IPDPS.2017.101

[33] Daniel Lemire and Leonid Boytsov. 2015. Decoding Billions of Integers per Second through Vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. doi:10.1002/spe.2203

[34] Christian Lemke, Kai-Uwe Sattler, Franz Färber, and Alexander Zeier. 2010. Speeding Up Queries in Column Stores: A Case for Compression. In *Data Warehousing and Knowledge Discovery (Lecture Notes in Computer Science, Vol. 6263)*. Springer, Berlin, Heidelberg, 117–129. doi:10.1007/978-3-642-15105-7_10

[35] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658. doi:10.14778/3007328.3007331

[36] Yihao Liu, Xinyu Zeng, and Huanchen Zhang. 2024. LeCo: Lightweight Compression via Learning Serial Correlations. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 65:1–65:28. doi:10.1145/3639320

[37] Yuyu Luo, Xuedi Qin, Nan Tang, and Guoliang Li. 2018. DeepEye: Towards Automatic Data Visualization. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 101–112. doi:10.1109/ICDE.2018.00019

[38] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 1633–1649. doi:10.1145/3318464.3389705

[39] LZ4 contributors. 2022. LZ4. https://github.com/lz4/lz4

[40] Antonio Mallia, Michal 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 (CIKM '19)*. ACM, 2193–2196. doi:10.1145/3357384.3358067

[41] Meta Platforms, Inc. and contributors. 2022. Zstandard. https://github.com/facebook/zstd

[42] mlcourse.ai. 2023. mlcourse.ai. https://github.com/Yorko/mlcourse.ai/tree/main/data

[43] C. G. Nevill-Manning and I. H. Witten. 1997. Identifying Hierarchical Structure in Sequences: A Linear-Time Algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82. doi:10.1613/jair.374

[44] 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 International Workshop on Data Management on New Hardware (DaMoN '25)*. ACM, Article 10, 10 pages. doi:10.1145/3736227.3736240

[45] NVIDIA Corporation. 2025. nvCOMP: High-Performance GPU Compression Library. https://developer.nvidia.com/nvcomp

[46] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 237–252. doi:10.1007/978-3-642-10424-4_17

[47] OpenStreetMap. 2017. OpenStreetMap Public Dataset. https://console.cloud.google.com/marketplace/product/openstreetmap/geo-openstreetmap

[48] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, Vienna, Austria, 1138–1149. doi:10.5555/1325851.1325979

[49] Shao-Jie Qiao, Han-Lin Fan, Nan Han, Lan Du, Yu-Han Peng, Rong-Min Tang, and Xiao Qin. 2025. Learning Database Optimization Techniques: The State-of-the-Art and Prospects. *Frontiers of Computer Science* 19, 12 (2025), 1912612. doi:10.1007/s11704-025-41116-7

[50] Eyal Rozenberg and Peter Boncz. 2017. Faster across the PCIe Bus: A GPU Library for Lightweight Decompression: Including Support for Patched Compression Schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware (DaMoN '17)*. ACM, New York, NY, USA, Article 8, 5 pages. doi:10.1145/3076113.3076122

[51] 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 (SIGMOD '20)*. ACM, 1617–1632. doi:10.1145/3318464.3380595

[52] 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 (SIGMOD '22)*. ACM, 1390–1403. doi:10.1145/3514221.3526132

[53] Claude Elwood Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27, 3 (1948), 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x

[54] Jian Shen, Ze Wang, David Wang, Jeremy Shi, and Steven Chen. 2019. Introducing AresDB: Uber's GPU-Powered Open Source, Real-Time Analytics Engine. https://www.uber.com/blog/aresdb/

[55] Evangelia A. Sitaridi, René Müller, Tim Kaldewey, Guy M. Lohman, and Kenneth A. Ross. 2016. Massively-Parallel Lossless Data Decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 242–247. doi:10.1109/ICPP.2016.35

[56] Larry H. Thiel and H. S. Heaps. 1972. Program Design for Retrospective Searches on Large Data Bases. *Information Storage and Retrieval* 8, 1 (1972), 1–20. doi:10.1016/0020-0271(72)90024-1

[57] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, and Franck Cappello. 2020. cuSZ: An Efficient GPU-Based Error-Bounded Lossy Compression Framework for Scientific Data. *CoRR* abs/2007.09625 (2020). doi:10.48550/arXiv.2007.09625

[58] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. 2021. Revisiting Huffman Coding: Toward Extreme Performance on Modern GPU Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 881–891. doi:10.1109/IPDPS49936.2021.00097

[59] Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron E. Lefohn. 2023. Random-Access Neural Compression of Material Textures. *ACM Transactions on Graphics* 42, 4 (2023), 88:1–88:25. doi:10.1145/3592407 Proceedings of SIGGRAPH 2023.

[60] Sebastiano Vigna. 2013. Quasi-Succinct Indices. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*. ACM, 83–92. doi:10.1145/2433396.2433409

[61] Ren Xuejun, Fang Dingyi, and Chen Xiaojiang. 2011. A Difference Fitting Residuals Algorithm for Lossless Data Compression in Wireless Sensor Nodes. In *2011 IEEE 3rd International Conference on Communication Software and Networks*. 481–485. doi:10.1109/ICCSN.2011.6013638

[62] Ren Xuejun and Ren Zhongyuan. 2018. A Sensor Node Lossless Compression Algorithm Based on Linear Fitting Residuals Coding. In *Proceedings of the 10th International Conference on Computer Modeling and Simulation (ICCMS '18)*. Association for Computing Machinery, New York, NY, USA, 62–66. doi:10.1145/3177457.3177482

[63] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proceedings of the VLDB Endowment* 6, 10 (2013), 817–828. doi:10.14778/2536206.2536210

[64] Feng Zhang, Zheng Chen, Chenyang Zhang, Amelie Chi Zhou, Jidong Zhai, and Xiaoyong Du. 2021. An Efficient Parallel Secure Machine Learning Framework on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2262–2276. doi:10.1109/TPDS.2021.3059108

[65] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, 1655–1669. doi:10.1145/3514221.3526130

[66] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient Document Analytics on Compressed Data: Method, Challenges, Algorithms, Insights. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1522–1535. doi:10.14778/3236187.3236203

[67] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Zwift: A Programming Framework for High Performance Text Analytics on Compressed Data. In *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 195–206. doi:10.1145/3205289.3205325

[68] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text Analytics Directly on Compression. *The VLDB Journal* 30, 2 (2021), 163–188. doi:10.1007/s00778-020-00636-3

[69] Feng Zhang, Chenyang Zhang, Jiawei Guan, Qiangjun Zhou, Kuangyu Chen, Xiao Zhang, Bingsheng He, Jidong Zhai, and Xiaoyong Du. 2025. Breaking the Edge: Enabling Efficient Neural Network Inference on Integrated Edge Devices. *IEEE Transactions on Cloud Computing* 13, 2 (2025), 694–710. doi:10.1109/TCC.2025.3559346

[70] Huanchen Zhang, Xiaoxuan Liu, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Order-Preserving Key Compression for In-Memory Search Trees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 1601–1615. doi:10.1145/3318464.3380583

[71] Jiaoyi Zhang and Yihan Gao. 2022. CARMI: A Cache-Aware Learned Index with a Cost-Based Construction Algorithm. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2679–2691. doi:10.14778/3551793.3551823

[72] Amelie Chi Zhou and Bingsheng He. 2014. Transformation-Based Monetary Cost Optimizations for Workflows in the Cloud. *IEEE Transactions on Cloud Computing* 2, 1 (2014), 85–98. doi:10.1109/TCC.2013.2297928

[73] Marcin Zukowski, Sándor Heman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. IEEE Computer Society, Atlanta, GA, USA, 59. doi:10.1109/ICDE.2006.150

Received October 2025; revised January 2026; accepted February 2026