The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code

Azim Afroozeh
Centrum Wiskunde & Informatica
The Netherlands

Peter Boncz
Centrum Wiskunde & Informatica
The Netherlands

ABSTRACT

The open-source FastLanes project aims to improve big data formats, such as Parquet, ORC and columnar database formats, in multiple ways. In this paper, we significantly accelerate decoding of all common Light-Weight Compression (LWC) schemes: DICT, FOR, DELTA and RLE through better data-parallelism. We do so by re-designing the compression layout using two main ideas: (i) generalizing the *value interleaving* technique in the basic operation of bit-(un)packing by targeting a virtual 1024-bits SIMD register, (ii) reordering the tuples in all columns of a table in the same Unified Transposed Layout that puts tuple chunks in a common "04261537" order (explained in the paper); allowing for maximum independent work for all possible basic SIMD lane widths: 8, 16, 32, and 64 bits.

We address the software development, maintenance and future-proofness challenges of increasing hardware diversity, by defining a virtual 1024-bits instruction set that consists of simple operators supported by all SIMD dialects; and also, importantly, by scalar code. The interleaved and tuple-reordered layout actually makes scalar decoding faster, extracting more data-parallelism from today's wide-issue CPUs. Importantly, the scalar version can be fully auto-vectorized by modern compilers, eliminating technical debt in software caused by platform-specific SIMD intrinsics.

Micro-benchmarks on Intel, AMD, Apple and AWS CPUs show that FastLanes accelerates decoding by factors (decoding >40 values per CPU cycle). FastLanes can make queries faster, as compressing the data reduces bandwidth needs, while decoding is almost free.

PVLDB Reference Format:
Azim Afroozeh and Peter Boncz. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

Analytical data systems routinely employ columnar storage. This allows queries to skip columns that they do not need, saving network, disk and memory bandwidth. Further, columnar storage tends to be more compact than row storage, thanks to *compression*.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Vectorized execution is a broadly adopted design for query execution where computational work in query expressions is performed on chunks of e.g., 1024 values called "vectors", by an expression interpreter that invokes pre-compiled functions that perform simple actions in loops over these vectors (arrays), thus amortizing function call overhead over 1024 tuples and allowing compilers to optimize these functions using techniques like loop-pipelining, code motion and auto-vectorization: generation of SIMD instructions [5].

Vectorized decoding carries over these efficient properties when applied to decoding compressed data. We focus on FOR, DICT, DELTA and RLE (resp. the Frame Of Reference [9], Dictionary, Delta and Run Length encodings). Also, when a vectorized table scan decompresses a vector, (compact) compressed data in RAM gets decompressed into an uncompressed vector, which is a small array of 1024 values, that fits the CPU L1/L2 caches and is immediately processed by the query pipeline, so it typically does not spill to RAM. As such, decompression happens between RAM and CPU, reducing memory, network and disk bandwidth consumption [39].

Parquet [1] also uses columnar encodings, albeit using a scheme that always applies DICT and represents the dictionary codes in variable-sized runs using bit-packing or RLE. Such variable-sized adaptivity hinders fast vectorized decoding [3], and the non-interleaved bit-packing and classic RLE it uses do not expose the opportunities for data-parallelism introduced by our techniques.

Compressed execution. We think scans in next-gen database systems should not decompress columns eagerly to their SQL type, which often is a wide integer (e.g., a decimal stored in 64-bits), but rather to the smallest type that makes the values processable by query operators. Modern systems like Procella [6], Velox [20] and DuckDB [25] support *compressed vector*s, where data is both randomly accessible yet still partially compressed: e.g., a FOR-vector or a DICT-vector, where 1024 values are represented as `uint8[1024]`, accompanied by one `uint64` base (FOR), resp. a pointer to a Dictionary. Such tight representations unlock optimizations (e.g., SIMD) for operators higher in a pipeline, and reduce the size of data structures, lessening (cache) memory pressure. It also causes *best case* scan decoding performance, where one decompresses a vector to its smallest possible lane-width, to become the *common case*.

FastLanes is a project initiated at CWI, intended as a foundation for next-generation big data formats. It introduces a new layout for compressed columnar data that increases the opportunities for data-parallel decoding, improving performance by factors. It does so in a way that works across the heterogeneous and evolving Instruction Set Architectures (ISAs) landscape, is future-proof, and minimizes technical debt by relying on scalar-only code.

1.1 Challenges and Contributions

In the FastLanes project we are re-designing columnar storage to expose more independence in data decoding, to make future query engines better at exploiting data-parallelism present in modern hardware. We contribute solutions to six challenges in Table 1:

Many SIMD widths. In the course of 25 years, SIMD ISAs have widened by a factor 8. Rather than taking the current widest SIMD ISA and proposing a data layout optimized for it, we preempt further widening of SIMD registers and propose a layout optimized for a **virtual 1024-bits register** `FLMM1024` that gets the best performance out of any existing ISA, and even from scalar code. At the lowest level of bits, this means FastLanes applies an *interleaved* bit-packed layout to 1024 bits; which distributes all logically subsequent e.g., 3-bit values round-robin over 128 separate 8-bit lanes. On the implementation level, it leads to vectorized decoding functions that deliver a vector of 1024 tuples at-a-time, in sometimes as little as 17 CPU cycles (an astonishing 70 values per CPU core cycle).

Heterogeneous ISAs. In order to deal with concurrently existing generations of x86 SIMD hardware, as well as ARM, where AWS Graviton1-3 and Apple M1-2 support 128-bits NEON, and Graviton3 also supports SVE; and other ISAs for POWER and RISC-V, we define a **simple instruction set**[1] on `FLMM1024` that is easily supported by the common denominator of all SIMD instruction sets. While it is out of scope in this paper, we think `FLMM1024` instructions on the FastLanes layout can also map efficiently to GPUs and other future data-parallel hardware (such as TPUs).

Decoding dependencies. Decoding RLE has an intrinsic control-dependency, as it needs a loop for emitting repeated values; but SIMD does not support control-instructions. DELTA decoding has an intrinsic data-dependency between subsequent values, which in SIMD are located in adjacent lanes; yet instructions with lane-dependencies are much slower. We tackle the latter problem by **reordering the column** using a technique we call "transposing", such that all lanes handle completely independent DELTA sequences. We then remap RLE to a combination of DELTA and DICT encoding, that leverages this very efficient DELTA decoding kernel.

Layouts that depend on lane-width. Previous work [15, 16, 21, 22, 27, 29, 31, 37] studied data encodings in isolation, but here we also look at the system context, i.e. table scans of multiple columns. When the optimal layout depends on a specific lane-width (8, 16, 32, 64 bits), this is problematic in that context. In table formats, different columns will store different value distributions which get bit-packed using different bit-widths and get decoded into types that fit different lane-widths. Our idea of transposing also runs into problems in this regard. Naively applied, it would lead to different column reorderings inside the same table. Therefore, we invented a very specific reordering of 1024 tuples that suits all possible lane-widths. This we call the **Unified Transposed Layout**. The gist of this reordering is to organize 1024 values in eight 8x16 transposed blocks, and to put these eight blocks in the order "04261537". We will explain why this order works well with any column-width.

[1]The idea is similar to [32] but as SIMD width interacts with data layout, we design for a concrete 1024-bits width. Rather than trying to cover all ISAs in intrinsics, our simple `FLMM1024` instruction set has a scalar implementation that gets auto-vectorized.

Challenge	FastLanes Solution
many SIMD widths	target a virtual FastLanes `FLMM1024` SIMD register
heterogenous ISAs	`FLMM1024` uses simple operators, present in all ISAs
decoding dependencies	reorder (transpose) columns to break dependencies
1 layout per lane-width	same Unified Transposed Layout forall lane-widths
keeping code portable	no intrinsics: use scalar code & auto-vectorization
LOAD/STORE-bound	vectorized execution & fused unpacking+decoding

Table 1: Challenges to efficient data-parallel decompression in big data formats, and how FastLanes tackles them.

Keeping code portable. The simple design of the `FLMM1024` Fastlanes 1024-bits instruction set allows to implement it in scalar code that uses `uint64` registers and operations. This portability also allows low-end CPUs that do not support any SIMD and that may even have 32-bits registers and memory addressing (but where compilers emulate 64-bits arithmetic) to also run FastLanes rather *efficiently* to their standard. On 64-bits CPUs, scalar FastLanes code achieves SIMD-like acceleration when handling small lane-widths (i.e. 8-bits gets 8x faster using 64-bits scalar). We find it remarkable that SIMD-friendly ideas like interleaving and transposing accelerate our scalar code, rather than slow it down. Last but not least, modern compilers can auto-vectorize our scalar code-path without loss of performance, **avoiding the need for SIMD intrinsics**, thus reducing technical debt and further making FastLanes future-proof.

Avoid getting LOAD/STORE-bound. We propose to use FastLanes decoding in vectorized execution, where the compressed data is read from RAM and gets decoded into 1024-value arrays, which are then processed from the CPU caches by the query pipeline. This reduces memory traffic by the compression ratio (often 2-3x). Further, most CPU time will be spent on the operators in the query pipeline, so scans run at much lower than the maximum decoding speed, further reducing bandwidth pressure. Sequential scans will trigger memory hardware prefetching, so good throughput can be reached. All this reduces the probability to be LOAD bound.

However, as FastLanes decoding is much faster than previous LWC schemes, and can achieve astonishing speeds, the decoding functions can become STORE bound, even when storing just into L1 cache. We show that **fusing our bit-unpacking kernels with the decoding kernels** for FOR/DELTA/RLE/DICT benefits performance, as this saves an intermediate STORE+LOAD.

1.2 Outline

The remainder of the paper is organized as follows. In Section 2 we explain these contributions in more detail, helped by a series of figures in visual language. First we explain 1024-bits interleaved bit-unpacking. The Unified Transposed Layout of FastLanes is motivated and explained around DELTA decoding. We further discuss efficient decoding of RLE exploiting this foundation. We follow-up in Section 3 with an evaluation of decompression performance of FastLanes bit-unpacking and DELTA and RLE decoding on all major hardware platforms. We also perform an end-to-end query execution benchmark based on Tectorwise [12] showing that using FastLanes decoding, instead of just an uncompressed in-memory array scan, can make a query faster. In Section 4, we discuss related work, covering the main differences between FastLanes and the state-of-the-art using both explanatory figures and micro-benchmarks. We conclude the paper and discuss future work in Section 5.

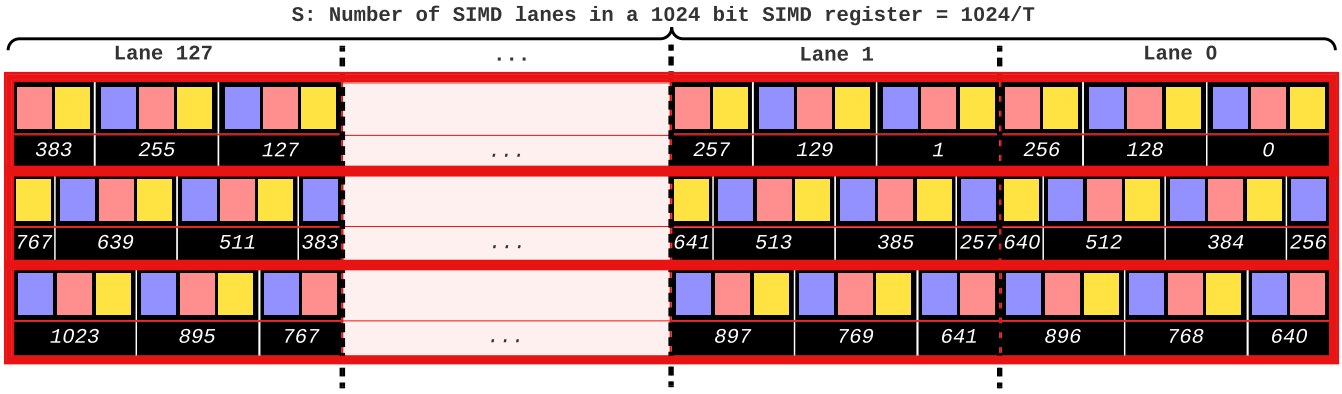


Figure 1: The 1024-bit interleaved layout. $B=3$ adjacent FLMM1024 words (red boxes, shown top-down) store 1024 values. Black bars indicate bit-packed values with their logical positions in the column: logically subsequent $W=3$-bit encoded values are round-robin spread into $S=128$ lanes of $T=8$-bits. In the first word, only the first two bits (yellow,pink) of the value at position 256 fit, so it is continued in the second word (blue bit). The value at position 640 is also split. This happens in all lanes.

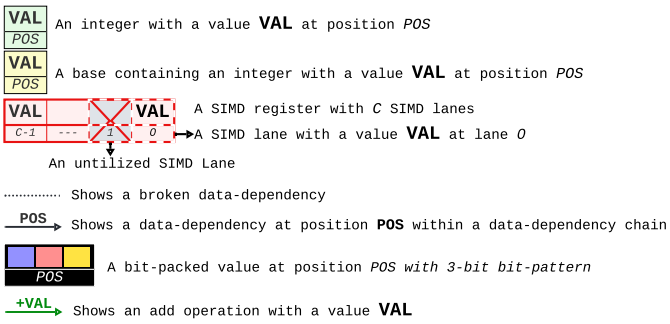


Figure 2: Legenda for our visual explanations.

2 FASTLANES

In order to explain the FastLanes compressed data layout, we make extensive use of drawings in the visual language introduced in Figure 2. We now explain the main FastLanes features in detail.

2.1 Many SIMD widths

Over the past three decades, SIMD register widths in x86 CPUs have doubled three times from MMX (64-bits) to SSE1-4 (128-bits 1999), AVX/AVX2 (256-bits, 2008) and AVX512 (512-bits, 2015). A next doubling is not imminent, but we do see GPUs - and Apple CPUs - adopting a 1024-bit cache-line, which facilitates such a move.

Existing SIMD decoding algorithms and their data layouts typically target a specific register width. Consider the *4-way* interleaved layout [16], which distributes bit-packed tuples among 4 SIMD lanes. This layout avoids expensive cross-lane `PERMUTE` or `BITSHUFFLE` instructions, needed if bits would be packed consecutively. While being efficient for unpacking four 32-bits values CPUs on 128-bit SIMD registers, this layout does not have enough parallelism for 256-bits or 512-bits registers. In response, the 8-way and 16-way interleaved formats were proposed [10], which are all different.

To preempt changing data formats when some ISA starts to support a wider SIMD register, FastLanes targets a still-not-existent register width, concretely 1024-bits.[2] One should note that as long as – expensive – lane-crossing operations are avoided, it is trivial to support data layouts designed for a wider register without performance penalty on a thinner SIMD register; just by using multiple identical thinner instructions working on adjacent data. The reverse is not true: supporting thin layouts on wide registers typically leads to lack of parallel work and unused lanes or expensive compensating actions such as `PERMUTE` and `BITSHUFFLE`.

Figure 1 shows the interleaved bit-packed layout in the example case of integers that can be encoded in 3 bits (W=3). To maximize decoding performance we use the smallest lane-width that fits that, i.e. 8-bits (T=8), and therefore we have 128 (S=1024/T=128) lanes in our `FLMM1024` word. Note that bit-packing is a building block that is used in all encodings and can optionally be combined with an exception-handling technique (such as "Patching" [39]), to handle - in this case – infrequently occurring values that do not fit 3 bits.

2.2 Heterogeneous ISAs

When new SIMD ISAs are introduced, we often see two kinds of asymmetries: (i) new operators that did not exist in a thinner ISA are introduced, or (ii) a wider register is introduced, but not all operators existing on thinner registers are (initially) supported on the wider register. Data layouts that depend on these operators are then problematic to support efficiently on all plausibly in-use hardware platforms, certainly for data systems that are distributed as binaries (pre-compiled).

Recently, ISA heterogeneity has significantly increased as ARM CPUs have become popular both on servers (AWS Graviton2,3) and with end-users such as data scientists (Apple M1,2); which bring their own subsets of NEON as well as SVE.

In order to support heterogeneous ISAs, FastLanes only uses simple operators, such as load/store, left/right-shift, and/or/xor, addition and set instructions; supported for all lane-widths, `T` $\in$ {8, 16, 32, 64} as shown in Listing 1. This instruction set can be trivially mapped to intrinsics in all previously mentioned thinner ISAs, just by using multiple identical instructions on independent

[2]We could have picked 2048 or 4096 as well; we chose to be conservative as the layout chunk-size grows with it: a chunk of 1024 W (bit-width) encoded values fit in exactly W `FLMM1024` registers. Larger chunk-sizes lead to worse compression ratios since the bit-width for bit-packing depends on the value-domain of a chunk (an exception mechanism to remove outliers can help to contain this problem). They also lead to an increased minimum vector-size, i.e. access granularity, imposed to the scan subsystem.

```
FLMM1024* // A pointer to 1024—bit word memory.
FLMM1024 // A variable of size 1024—bit

// Load 1024—bits from memory address ADR
FLMM1024 LOAD<T>(FLMM1024* ADR);

// Store 1024—bits from REG into memory address ADR
void STORE<T>(FLMM1024* ADR, FLMM1024 REG)

// forall T—bit lanes i in REG return (i & MASK) << N
FLMM1024 AND_LSHIFT<T>(FLMM1024 REG, uint<T> MASK, uint8 N)

// forall T—bit lanes i in REG return (i & (MASK << N)) >> N
FLMM1024 AND_RSHIFT<T>(FLMM1024 REG, uint<T> MASK, uint8 N)

// forall T—bit lanes (a,b) in (A,B) return (a & b)
FLMM1024 AND<T>(FLMM1024 A, FLMM1024 B)

// forall T—bit lanes (a,b) in (A,B) return (a | b)
FLMM1024 OR<T>(FLMM1024 A, FLMM1024 B)

// forall T—bit lanes (a,b) in (A,B) return (a ^ b)
FLMM1024 XOR<T>(FLMM1024 A, FLMM1024 B)

// forall T—bit lanes (a,b) in (A,B) return (a + b)
FLMM1024 ADD<T>(FLMM1024 A, FLMM1024 B)

// forall T—bit lanes, return VAL.
FLMM1024 SET<T>(uint<T> VAL)
```

Listing 1: FastLanes simple SIMD instruction set, with `FLMM1024` 1024-bits registers and `T`-bits lanes; `T` ∈ {8, 16, 32, 64}. It can be trivially mapped onto any existing SIMD ISA, as well as onto scalar code using `uint64`: ISAs with thinner registers just use multiple identical instructions on multiple registers and adjacent memory to reach 1024-bit width.

registers or adjacent memory locations, to reach the 1024-bit width of our virtual `FLMM1024` register. The extreme example of this is our Scalar_T64 code-path, which relies on 64-bits integers (`uint64`):

```
struct { uint64 val[16]; } FLMM1024; // 16*uint64 = FLMM1024
FLMM1024 AND<8>(FLMM1024 A, FLMM1024 B) {
  FLMM1024 R;
  for(int i=0; i<16; i++) R.val[i] = A.val[i] & B.val[i];
  return R;
}
```

As a detail, we note that we combined the shift instructions with AND functionality. In bit-packing, these two operations are typically followed by each other anyway, so in those cases, the combined instruction is a shorthand. Another reason to introduce this shorthand is our Scalar_T64 code-path that manipulates `uint64` values. As shown above, we can support for instance eight 8-bits lanes using instructions on `uint64`. However, shift instructions on `uint64` could transport bits from one lane into another, something that is guaranteed not to happen in SIMD instructions. But, by performing the AND before shifting in such a way that bits that would cross a lane are masked out, this problem can be prevented by manipulating the (constant) mask value, at no additional cost.[3]

Listing 2 shows the implementation for unpacking 3-bit (W=3) codes into 8-bit (T=8) integers. Rather than writing such code by hand, we generate it statically for all $1 \leq W \leq 64$, $T \in \{8,16,32,64\}$

[3]Note that cross-lane bit-spilling is also a risk in the ADD operator. However, as SIMD ISAs do not support overflow detection, usage of SIMD ISAs for summations already requires the use of overflow prevention techniques in order to ensure correctness. Hence for ADD we can assume that overflow does not happen.

```
uint<8> MASK1 = (1<<1)—1, MASK2 = (1<<2)—1, MASK3 = (1<<3)—1;
FLMM1024 r1, r0;
r0 = LOAD<8>(in+0);
r1 = AND_RSHIFT<8>(r0,0,MASK3); STORE<8>(out+0,r1);
r1 = AND_RSHIFT<8>(r0,3,MASK3); STORE<8>(out+1,r1);
r1 = AND_RSHIFT<8>(r0,6,MASK2);
r0 = LOAD<8>(in+1);             STORE(out+2,OR<8>(r1,
     AND_LSHIFT<8>(r0,2,MASK1)));
r1 = AND_RSHIFT<8>(r0,1,MASK3); STORE<8>(out+3,r1);
r1 = AND_RSHIFT<8>(r0,4,MASK3); STORE<8>(out+4,r1);
r1 = AND_RSHIFT<8>(r0,7,MASK1);
r0 = LOAD<8>(in+2);             STORE(out+5,OR<8>(r1,
     AND_LSHIFT<8>(r0,1,MASK2)));
r1 = AND_RSHIFT<8>(r0,2,MASK3); STORE<8>(out+6,r1);
r1 = AND_RSHIFT<8>(r0,5,MASK3); STORE<8>(out+7,r1);
```

Listing 2: Interleaved bit-unpacking kernel in `FLMM1024` SIMD for T=8 and W=3. We use code-generation to create such implementations for all combinations of T and W ($W < T$).

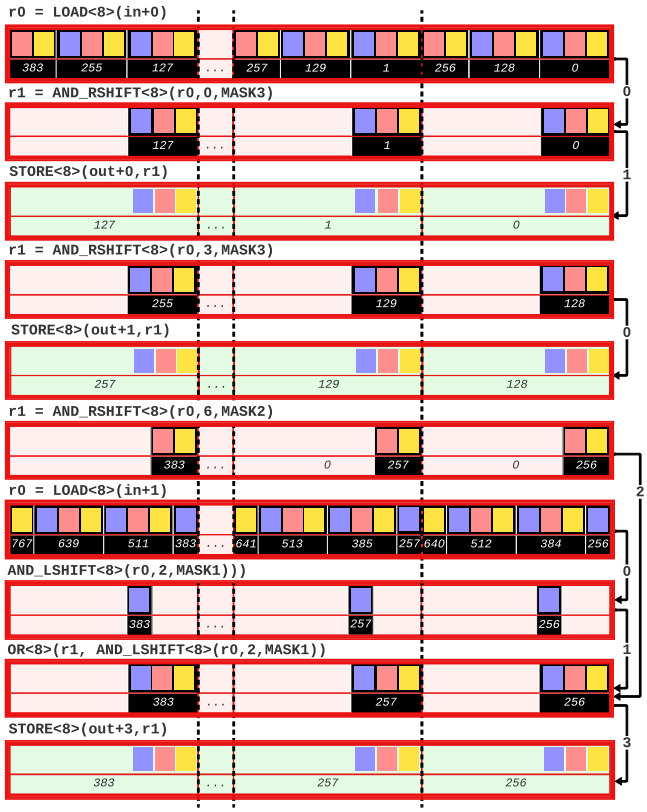
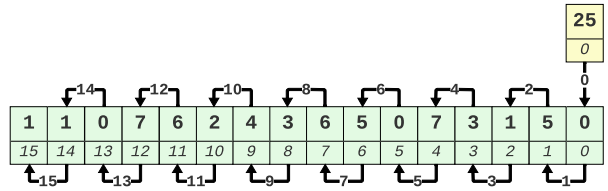
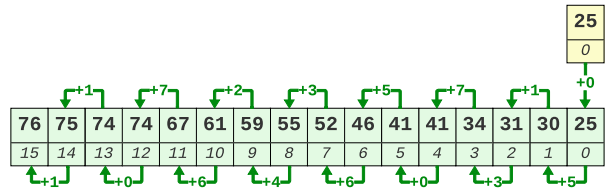


Figure 3: Lines 3-8 of Listing 2 in action: ten `FLMM1024` instructions bit-unpack the first 384 3-bits codes into 8-bit integers. The investment in interleaving of bits leads to perfectly sequential unpacked integers using few simple instructions.

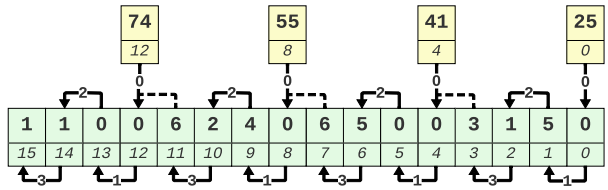
where $W < T$ (116 pre-compiled functions that each deliver a vector of 1024 values). Figure 3 shows the algorithm in action: in 10 instructions, 384 values are unpacked. On this unpack kernel, Intel AVX512 CPUs get to the astonishing speed of 70 values per cycle = 140 billion values per second on one 2GHz core. Given 3-bits per value this requires 52GB/s - close to RAM bandwidth limit. In reality, however, a query pipeline spends at least a few cycles per value in its operators, so the pipeline runs 100x slower; but with this unpacking speed the decompressing scan is practically free.



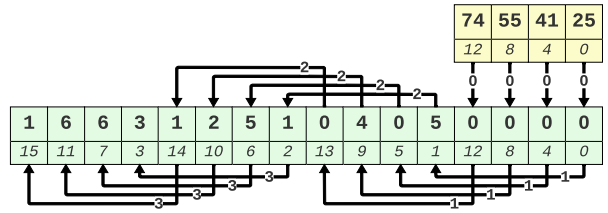
(a) Default DELTA layout with data dependencies on arrows.



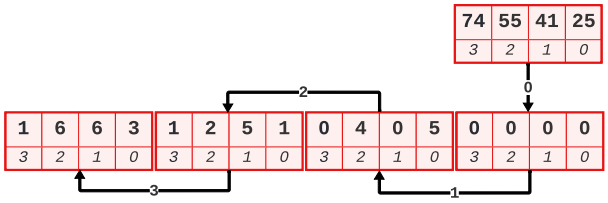
(b) The process of decoding DELTA-encoded data (green arrows).



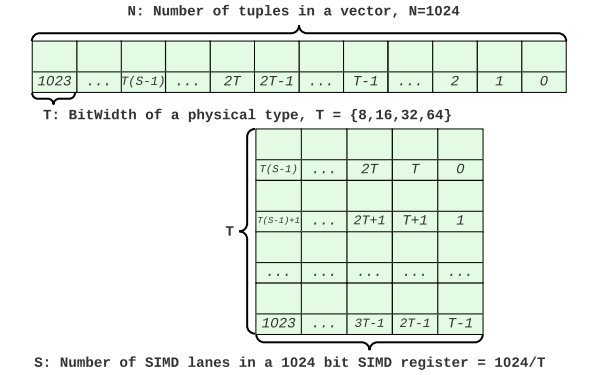
(c) Example DELTA layout with multiple bases.



(d) Transposed data layout.



(e) SIMD-friendly DELTA decoding on the transposed layout.



(f) Transposed Layout: value order depends on widths S & T

Figure 4: The Transposed Data Layout. Idea: reorder column values to make data dependencies SIMD-friendly.

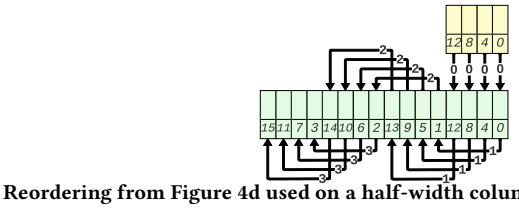
2.3 Dealing with Sequential Data Dependencies

Dependencies between subsequent values are SIMD-unfriendly since adjacent values end up in adjacent lanes. Figure 4a shows that the default layout (one value after the other) has this problem. The additions needed for DELTA decoding are lane-crossing operators: suppose the values in in Figure 4b are 32-bits, then adding the values at position 0 and position 1 correspond to different lanes (if e.g., positions 0-3 were loaded in a 128-bit SIMD register).

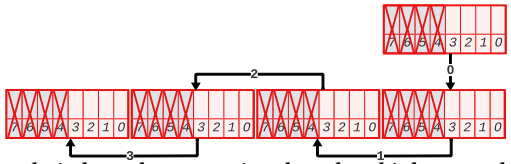
In these figures, the yellow boxes indicate *base* values. These bases provide entry-points to start DELTA decoding. In FastLanes, we allow to start decoding with a granularity of 1024 tuples. Base values would be found in the header of a compressed columnar block. But, rather than having one base per vector, Figure 4c shows the idea of having four bases. This allows to start decoding at positions 0,4,8 and 12. It still does not solve the lane-crossing problem, though. Figure 4d shows the "transposed" layout, that **stores the values out-of-order**. The order for the first 16 values here is 0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15. Figure 4e show this leads to optimal 128-bits SIMD processing: only 4 additions are needed.

We call this re-ordering a transposition because the idea is to cut up the value column in SIMD register-sized chunks and put these chunks vertically under each other, as shown in Figure 4f. In case of our 1024-bits `FLMM1024` register, this means that this matrix has exactly T rows and S columns; where T is the value (=lane) bit-width and S is the amount of such values in a register.

We argue that changing the tuple order is not problematic in the database scan context. Relational algebra is set-based and query operator semantics typically do not depend on order, so if the tuples arrive perturbed from insertion order, they can usually be processed in whatever order they arrive. Even if the order matters for the query result or operator semantics, the original order could be restored or encoded in a *selection vector*. While the presence of a selection vector can slow down operations, it can often be avoided: vectorized query executors typically have an optimization where simple arithmetic operators (that cannot raise errors) will ignore (identical) selection vectors on all parameters, if many tuples are still in play, executing the operation on all values, at much lower per-value cost thanks to full sequential access (and SIMD).

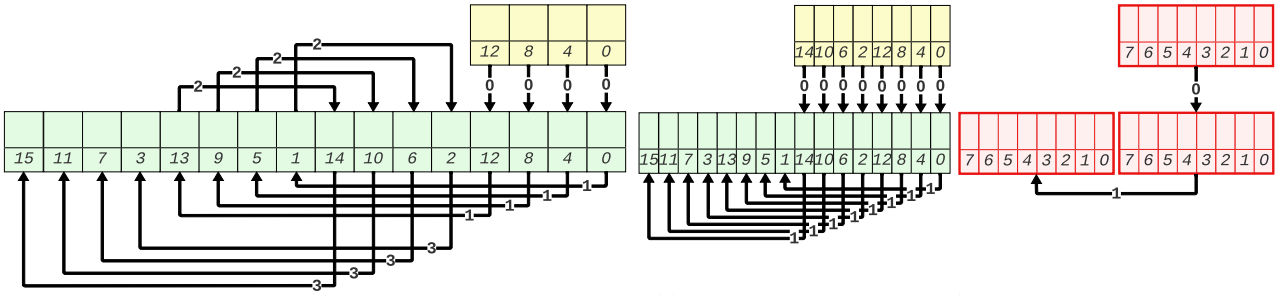


(a) Reordering from Figure 4d used on a half-width column

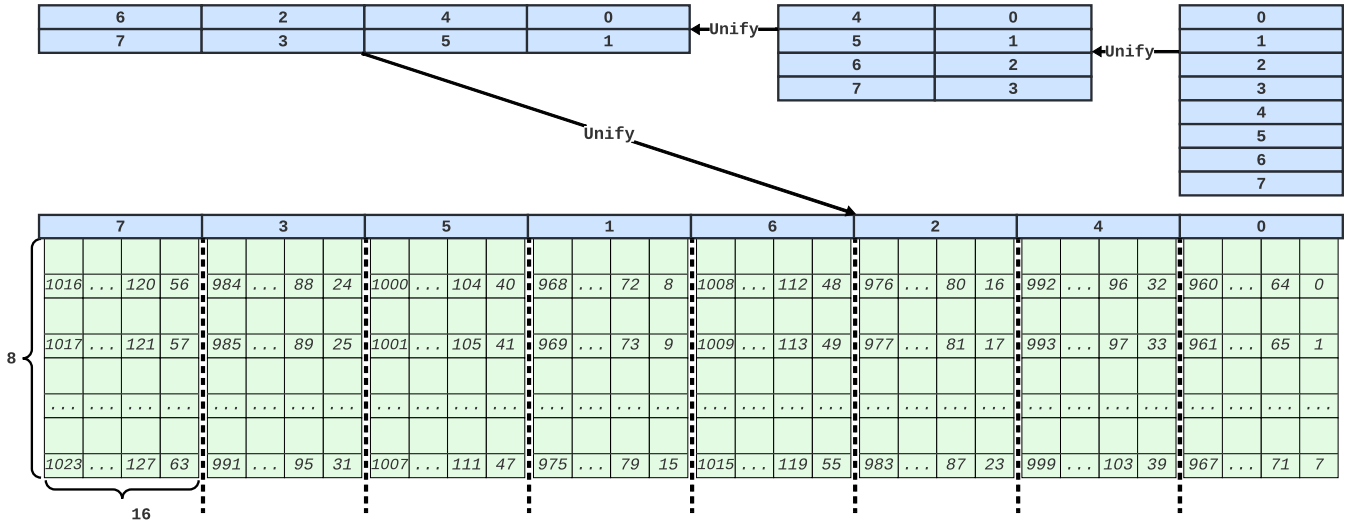


(b) We need 8 independent operations here, but this layout only offers 4. This leads to unused SIMD lanes while decoding.

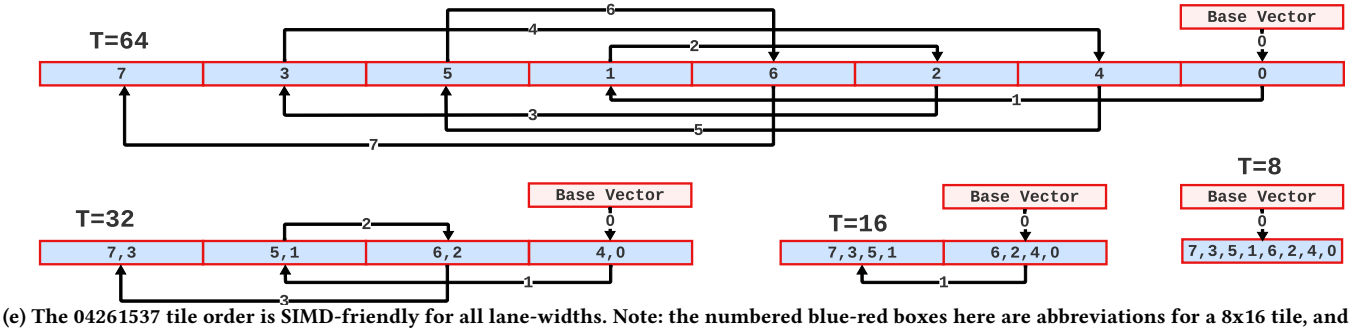
Figure 5: Transposed Layout and resulting value reordering designed for one data type, is unsuited for thinner data types.



(a) This reordering is also SIMD-friendly for the wide type, but.. (b) ..applied to the thin type.. (c) ..is also SIMD-friendly for that.



(d) From the 64-bits transposed layout with 8x1 tiles of 8x16 values (upper right), we unify to 4x2 tiles for 32-bits types, then to 2x4 for 16-bits and 1x8 for 8-bits, in 04261537 tile order. The final value order is 0,64,..,960,32,96,..,992,..,..,63,127,..,1023 (top-to-bottom, repeated right-to-left)



(e) The 04261537 tile order is SIMD-friendly for all lane-widths. Note: the numbered blue-red boxes here are abbreviations for a 8x16 tile, and imply 8 SIMD ADD operations each during DELTA decoding. This layout benefits all encodings with dependencies (i.e., also RLE).

Figure 6: Unified Transposed Layout: (a)-(c) idea of order unification, (d) how our unified approach arrives at the 04261537 order (blue) of 8x16 tiles (green) and the final value order (green), (e) how it provides data-parallelism for all possible lane-widths. Notably, FastLanes does not only store each sequence of 1024 tuples permuted in this reordering, but the individual columns are usually also encoded with some LWC scheme (DELTA, FOR, DICT, RLE), which involves bit-packing using 1024-bit interleaving (Figure 1). So the eventual bit-sequences stored are humanly hard to grasp. However, decoding the values requires only regular and astonishingly fast calculations that are completely data-parallel.

2.4 The Unified Transposed Layout

In our Transposed Layout, the order of the tuples depends on T. This creates a problem for database scans: relational tables consist of *multiple* columns and different columns will have different widths. However, when we reorder tuples, we should use the same order for all columns, because a scan needs to create a consistent stream of tuples.[4] Figure 5 shows that when we apply the reordering from Figure 4d to a data type of half the width, there is not enough independent work for the thinner type. In our example, the wide data-type was 32-bits such that 4 values fit a 128-bits SIMD register. So when putting a column of 16-bits integers in that order, we see that we only can take advantage of four lanes, instead of 8. In this case, the problem can be solved by just using a different ordering, shown in Figure 6a-c, that works well with columns of both widths.

Our Unified Transposed Layout provides a generic solution to this problem for all lane-widths. The basic building block are transposed tiles of 8x16 values. We have eight such tiles for each vector of 1024 tuples. For the widest 64-bits type, each row in the tile is one `FLMM1024` register, making it a suitable format to process one tile-at-a-time: for DELTA decoding, the 8 rows are processed using 8 `FLMM1024` ADD<64>. In case of 32-bits values, however, one row occupies half a register, so we need to group two independently processable tiles together in one register. This is done by taking the lower half of tiles 0-7 and placing them to the left, arriving at 4 rows of 2 tiles. This process repeats for 16-bits and 8-bits, arriving at a single row of 8 tiles in the 04261357 ordering (blue). The complete value ordering for all 1024 tuples is shown in green.

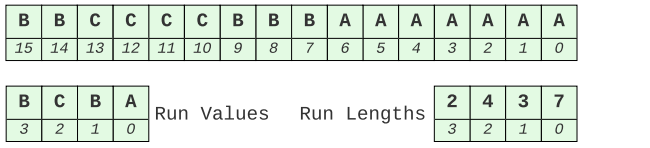
One can ask if 04261357 is the only ordering (starting at 0) that is suitable for DELTA decoding. We want to start at 0, because for 64-bits values we compute on data from one tile at-a-time, starting at tile 0; and for 64-bits data, the header thus holds bases for tile 0 only (see Figure 6a-b with base values in yellow). Beyond starting at 0, the second desirable property is that for processing tiles in SIMD operations, we need the subsequent operations to touch *directly subsequent* tile numbers in the same SIMD lane position.

Now the proof. Considering 16-bits values, where four tiles fit the SIMD register width, and given that 0 is first; we see that 1 must be in fourth position (as it must be subsequent in 0xxx⟶1xxx). In fact, the only way to get subsequent numbers in the two halves of the ordering is to have all even numbers first, and the odd numbers later. Now, considering 32-bits data types, where data from two tiles is processed at-a-time, the ordering should start with 04. Because, if we would start with 02, then after 02⟶13, the next SIMD operation should be on 24, but tile 2 was already processed. The other even choice 06 runs out of work, as after 06⟶17 there is no tile 8. As the first pair is 04, the third pair must be 15, and this fixes the second pair to 26 and the final pair to 37; so we arrive at 04261537 as the only ordering with the desired properties. Figure 6e shows that for 8-bits types, DELTA decoding processes: *bases* ⟶ 04261537 (drawn, as all layouts, right-to-left in our Figures). For 16-bits types the processing order is: *bases* ⟶ 0426 ⟶ 1537. For 32-bits it is: *bases* ⟶ 04 ⟶ 15 ⟶ 26 ⟶ 37. For 64-bits: *bases* ⟶ 0 ⟶ 1 ⟶ .. ⟶ 7.

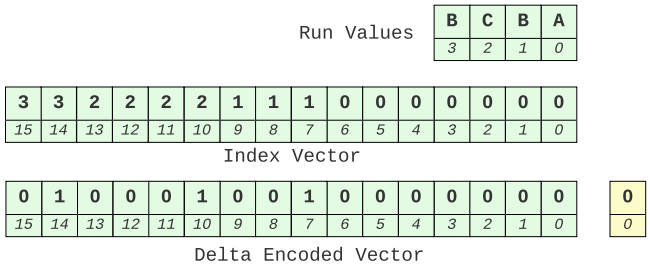
[4]Even if a query processor would be able to work with column vectors that each have a different value order, e.g., by accompanying each with their own selection vector that restores order; this would likely carry performance penalties due to the indirect memory access needed and reduce the applicability of our format to systems that could do this. Therefore we enforce the ability to retrieve all column data in the same order.

FastLanes-RLE. Value sequences get Run Length Encoded in classic RLE as *(value,length)* tuples. Decoding requires two nested loops: one that iterates over the tuples, and inside, one that iterates over *length*; while writing out the *value*-s. A loop is by definition scalar, and the inner loop will suffer from branch mispredictions on short lengths. The best SIMD acceleration so far for RLE works when run-lengths are large, such that the uncompressed run is very significantly larger than the SIMD register. In this case, one can set all lanes of a SIMD register to the constant *value*, and reduce the amount of STORE instructions by the amount of lanes [7].

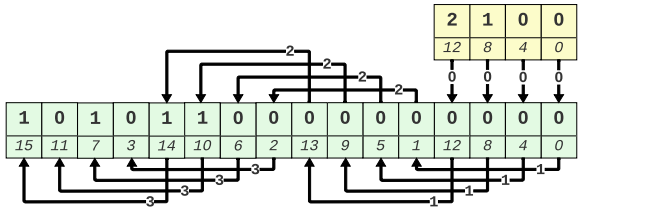
We propose a new scheme called Fastlanes-RLE, that maps RLE to DELTA and supports storage reordered in the Unified Transposed Layout. It targets systems like Velox [20] and DuckDB [25], that prefer to represent decoded RLE as compact in-flight Dictionary vectors; rather than full/eager decompressed vectors. The twist here is that the Dictionary is the Run Value vector from RLE, and hence may contain duplicates. The Index Vector monotonically increases by one, whenever a new run starts. FastLanes-RLE uses 16-bit indexes for vectors with many short runs and 8-bits otherwise. These Index Vectors are DELTA encoded using only 1-bit per value. Base storage in the 8-bit case can use 3-bit bit-packing, adding .375 bits of storage per value, making the compression ratio better than classic RLE, up to average run-lengths of 12. For longer average run-lengths, we should use 0-bit DELTA encoding, that memsets the Index Vector to 0, and where the 1-s are inserted by an exception mechanism (we will cover such mechanisms in follow-up work).



(a) A decompressed vector and its classic RLE representation as two vectors: Run Values and Run Lengths.



(b) FastLanes-RLE, and how its Index Vector is DELTA encoded.



(c) FastLanes-RLE reorders the Index Vector in Unified Transposed Layout: compatible with other columns and enabling fast decoding.

Figure 7: FastLanes-RLE: a fast and compact encoding scheme targeting in-flight partially compressed vectors [20, 33]

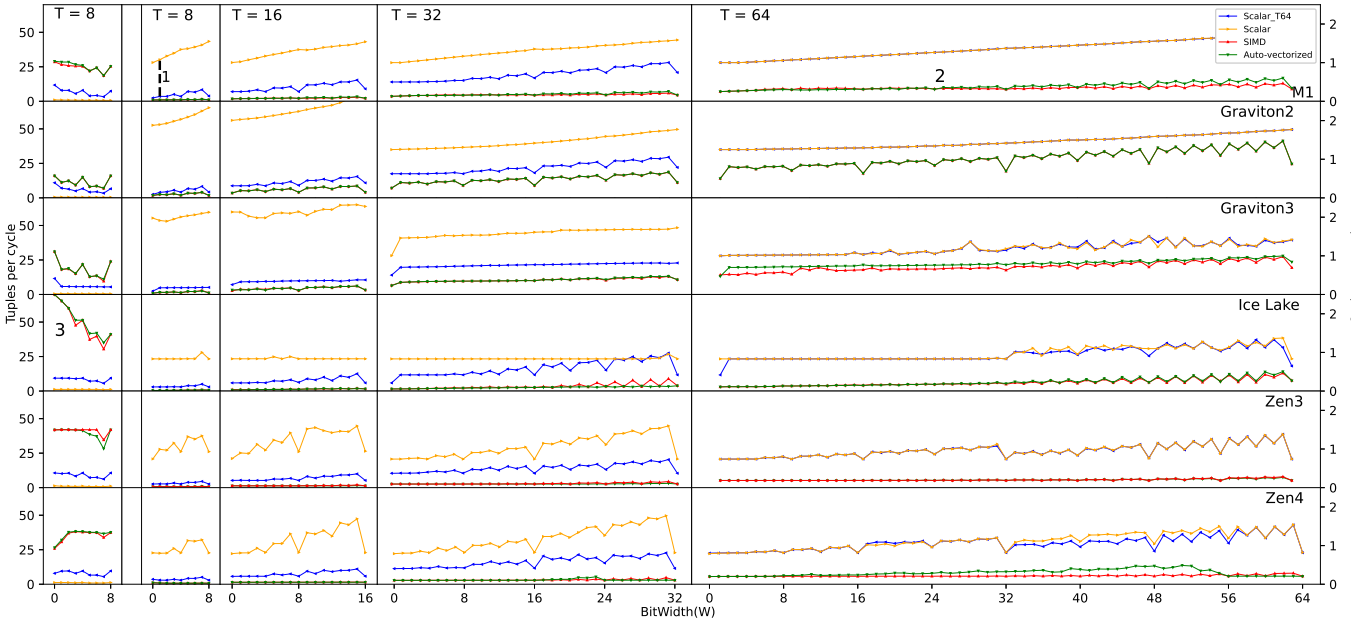


Figure 8: Bit-unpacking performance of the 1024-bit interleaved layout. (1) `Scalar_T64` uses 64-bit scalar registers as quasi-SIMD and beats naive `Scalar` up to 8x. (2) clang++ auto-vectorizes `Scalar` perfectly, matching performance of explicit SIMD intrinsics. (3) Decoding can reach 70 tuples/cycle (T=8, W=1). Except in the leftmost box here (tuples/cycle), lower is better in all Figures (cycles/tuple).

3 EVALUATION

The C++ FastLanes library is released under a MIT license in open source and will be put in github.com/cwida/FastLanes on Jan 7. We now experimentally evaluate the following questions:

- **(Q1)** What is the absolute speed of the proposed FastLanes 1024-bit interleaved bit-unpacking?
- **(Q2)** Does decoding performance scale with SIMD width, and how does it vary between the platforms listed in Table 2?
- **(Q3)** Can scalar code profit from 1024-bits interleaving and the Unified Transposed Layout?
- **(Q4)** What is the performance of the scalar implementation, and how well does compiler auto-vectorization compare with the use of explicit SIMD intrinsics?
- **(Q5)** How does the proposed Unified Transposed Layout influence decoding performance, specifically for LWC schemes with sequential dependencies, such as DELTA?
- **(Q6)** What effect on end-to-end query performance could the adoption of FastLanes have?

We also investigate the performance benefits of potentially fusing the implementations of bit-unpacking and decoding kernels. Note that in Section 4, we present additional micro-benchmarks while comparing FastLanes with related work.

3.1 Micro-benchmarks

We implemented bit-unpacking and decoding into $T = \{8, 16, 32, 64\}$ result columns in 4 different ways: `Scalar`, `Scalar_T64`, `SIMD`, and `Auto-vectorized`. The `Scalar` code unpacks/decodes one `uint`T value at-a-time. The `Scalar_T64` implementation treats a `uint64` variable as a quasi-SIMD register consisting of 64/T lanes of T-bits.

We used clang++ for our experiments. To make sure that our scalar code is not auto-vectorized, we explicitly disabled the auto-vectorizer for the `Scalar` and `Scalar_T` implementations by using: `-O3 -mno-sse -fno-slp-vectorize -fno-vectorize`.

Architecture	Scalar ISA	Best SIMD ISA	CPU Model	Frequency
Intel Ice Lake	x86_64	AVX512	8375C	3.5 GHz
AMD Zen3	x86_64	AVX2 (256-bits)	EPYC 7R13	3.6 GHz
AMD Zen4	x86_64	AVX512	Ryzen9 7950X	4.5 GHz
Apple M1	ARM64	NEON (128-bits)	Apple M1	3.2 GHz
AWS Graviton2	ARM64	NEON (128-bits)	Neoverse-N1	2.5 GHz
AWS Graviton3	ARM64	NEON (128-bits) SVE (variable)	modified Neoverse-V1	2.6 GHz

Table 2: Hardware Platforms Used

The `SIMD` implementations use explicit SIMD intrinsics. Note that for ARM64, all SIMD implementations are based on NEON instructions. This is because our experiments on Graviton3 showed that SVE [30] is slower than NEON. Finally, the `Auto-vectorized` implementation is the `Scalar` implementation, with the difference that auto-vectorization is not disabled.

These micro-benchmarks aim to characterize pure CPU cost and decompress a single vector 30M times; hence all data is L1 resident. We report CPU cycles per value (**lower is better!**), but for T=8 bit-unpacking also the reverse: values per cycle (cycles per value there get close to 0 and hard to discern). These measures make the results more meaningful to compare across platforms than elapsed time, as our hardware comes from different frequency classes (hi/mid/low end, consumer vs. server). We disabled CPU turbo scaling features where present to make clock normalization stable.

Bit-unpacking. Figure 9 we see that the 1024-bits interleaving of packed data does not even hinder `Scalar` decoding: performance is equal to the naive "horizontal" (non-interleaved) bit-packed layout. But, only the interleaved layout provides the opportunity of decoding multiple lanes in parallel seized by `Scalar_T64`, making it 8x faster than `Scalar` on 8-bits values. As for **(Q1)**, Figure 8 shows the high speed of FastLanes decoding: thanks to SIMD it significantly outperforms `Scalar` across all platforms: 40x-70x for 8-bits, to 3x-4x for 64-bits types. Regarding **(Q2)**: we do see that Gravitons

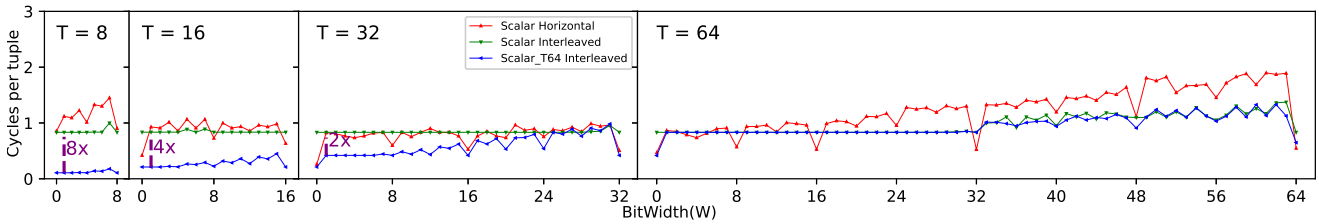


Figure 9: horizontal vs. 1024-bit interleaved. `Scalar` **bit-unpacking performance with 1024-bits interleaving is equal to the naive horizontal layout (red=blue). The bit-interleaving approach allows** `Scalar_T64` **(green) to get up to 8x faster (Ice Lake).**

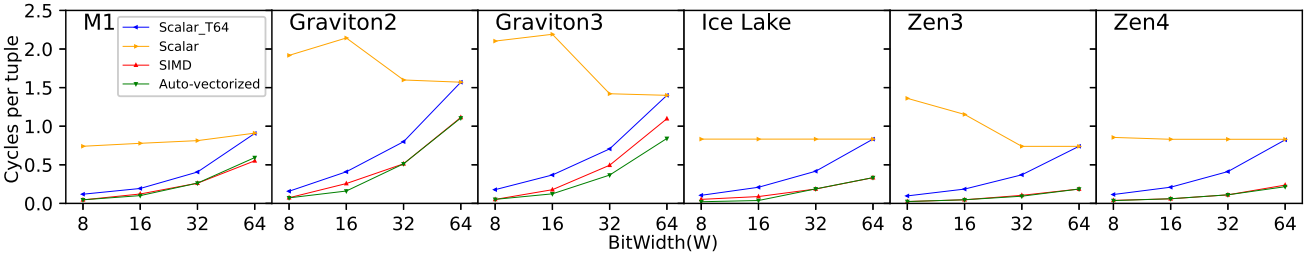


Figure 10: FastLanes DELTA decoding, for all bit-widths & platforms: very high performance for `Auto-vectorized`**. Also,** `Scalar_T64` **profits from data-parallelism in the Unified Transposed Layout, whereas** `Scalar` **cannot and can be >40x slower than** `SIMD`**.**

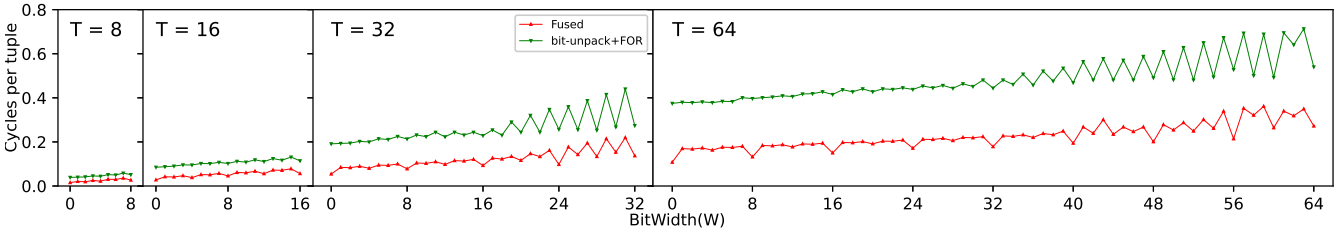


Figure 11: Fusing 1024-bits interleaved bit-unpacking with decoding (FOR) improves performance (Ice Lake).

have weaker SIMD; which especially shows for 64-bits types. Apple M1 also has just 128-bit NEON, but clearly has more instruction level paralellism (ILP). Wider SIMD does not always equate more performance: despite supporting AVX512, Zen4 is not faster than Zen3. This is expected if the CPU executes one AVX512 instruction using two AVX2 (256-bits) units. The absence of dependencies and the opportunities for data-parallelism that FastLanes code exposes, make it profit from total CPU execution capability, which is the product of ILP and register width.

Figure 8 highlights that (1) `Scalar_T64` is indeed $\frac{64}{T}$ times faster than Scalar for different Ts **(Q3)**; (2) clang++ can auto-vectorize our `Scalar` code, matching the performance of explicit intrinsics — denoted `SIMD` **(Q4)**; (3) FastLanes can decompress 70 tuples per cycle for 8-bits types **(Q1)**, where SIMD paralellism is maximal. Point (2) means that when incorporating FastLanes in future systems, we recommend just using the Scalar code paths; in fact for the kernels described in this paper, just the `Scalar_64` code is enough. This result significantly enhances the future-proofness of FastLanes.

Unified Transposed Layout. We performed experiments for **(Q5)** regarding DELTA decoding for all six hardware platforms. Figure 10 shows that the Unified Transposed layout – the idea to reorder the tuples in order to break sequential dependencies – also benefits our `Scalar_T64` code-paths, that uses `uint64` scalar registers as if they were 8x8-bits, 4x16-bits or 2x32-bits SIMD registers. In terms of scalar performance, M1 tops Ice Lake clock-for-clock. Remarkably, Graviton and Zen3 are slower in scalar additions on 8- and 16-bits numbers than on 32- and 64-bits. The Gravitons again show weak SIMD. Performance can again be very high, like >40 tuples per cycle on the faster platforms for 8-bits DELTA. Most DELTA decoding will be on the larger datatypes (32-, 64-bits), but FastLanes-RLE (evaluated later) uses very fast on 1-bit decoding in a 16-bits lane.

As bit-unpacking and FastLanes decoding use dependency-free instructions, column contents do not influence performance at all. Only the bit-width matters, hence we evaluate all bit-widths.[5]

Fusing Bit-packing and Decoding. The 116 bit-unpacking kernels we generate for all bit-packing widths W and unpacked type-widths $T \leq W$ could possibly be fused with the decoding kernel for DELTA, FOR, DICT and FastLanes-RLE in a single kernels that do both unpacking and decoding. The benefit of fusing is that the STORE instructions that bit-unpacking ends with, and the LOAD instructions that decoding starts with, are saved. Figure 11 shows that fusing indeed improves the decompression speed.

In case of decoding into compressed vectors, fusing is not needed for DICT and FOR (decoding is just bit-unpacking in that case – therefore we do not micro-benchmark these schemes separately). For decoding DELTA into a compressed FOR vector, we can use fusing; what is then needed is to keep MinMax stats per vector, and subtract Min from the bases before decoding.

[5]Regarding (ordered) DELTA columns, we finally argue that subsequent query performance after decompression is not likely to be affected even if the tuple order is left transposed, since the permutation caused by transposing is within a 1024-vector only, and hence localized, such that any column order is largely preserved.

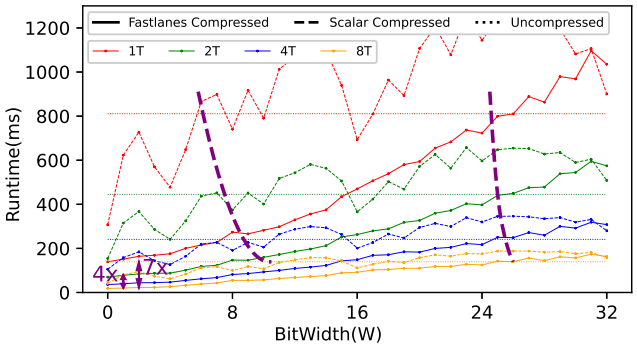


Figure 12: `SELECT SUM(COL) FROM TAB` runtime for various `COL` bit-widths and threads (T) on Ice Lake. The crossover point where decompressing scans (plots) outperform plain array scans (horizontal lines), moves from a minimal compression ratio of 4x (≈8bits) with `Scalar` decoding to just 25% compression (≈24bits) with FastLanes. Note that with higher thread counts, the crossover point (thick stripes) moves right a bit, as RAM bandwidth gets scarcer. FastLanes can then improve end-to-end performance up to 7x vs. uncompressed and 4x vs. scalar.

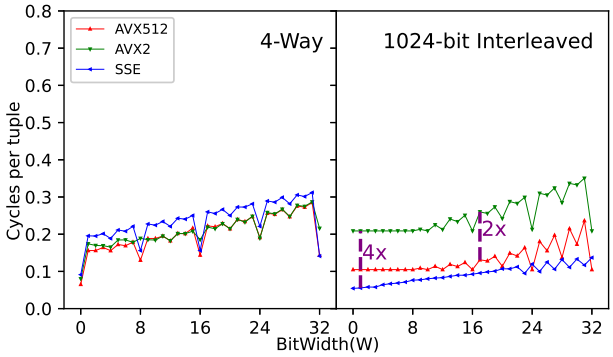


Figure 13: Bit-unpacking using the 4-way layout vs. 1024-bit interleaved layout, where $T = 32$ (Ice Lake). The 4-way layout cannot take advantage of wide SIMD registers, with a performance penalty of 2x resp. 4x for AVX2 resp. AVX512.

3.2 End-to-End Query Performance

We also ran a complete query pipeline, by integrating FastLanes in the experimental Tectorwise [12] vectorized query processor. We created a table `TAB` with a single column `COL` that has $10 * 2^{28}$ `uint32` integer values (10GB), and benchmarked the query `SELECT SUM(COL) FROM TAB` on our IceLake platform.

Figure 12 shows the performance of this query, depending on the domain of the values in the column, which is uniform-randomly generated from the domain $[0\text{-}2^W\rangle$. We run this unmodified Tectorwise query, that reads COL from an `uint32` array, and two modified versions (FastLanes and `Scalar`) that scan a compressed `COL` – which gets bit-packed in W bits per value. In all cases the data is RAM-resident. As for **(Q6)**, we thus see that reading from FastLanes typically makes a query **faster**, despite the decompression, because the query needs less RAM-bandwidth. Parallel execution increases the RAM bottleneck: with 8 threads we see up to 7x end-to-end performance improvement vs. uncompressed (and 4x vs. `Scalar`). FastLanes shifts the crossover point where queries get faster from data with a >4x compression ratio (`Scalar`) to almost any data.

4 RELATED WORK

For more than two decades, researchers have been trying to use SIMD instructions to improve the performance of database systems [14, 38]. Much of this effort has been made on SIMDizing the compression and decompression of data [15, 16, 21, 22, 27, 29, 31, 37]. Surveys of these SIMDized compression schemes are [3, 7].

Bit-packing. Zukowski *et al.* propose to bit-pack 128 integers sequentially using the same bit-width [39]. Schlegel *et al.* call this layout horizontal [27]. Willhalm *et al.* propose a SIMDized bit-unpacking for the horizontal layout [35]. In addition to the horizontal layout, Schlegel *et al.* propose the k-way vertical layout [27], where each of the k consecutive bit-packed values are distributed among consecutive memory words. This vertical idea is also called interleaved layout, and we use that terminology in this paper. This distribution allows to have bit-packed values in different SIMD lanes and avoids the extra `PERMUTE` instruction, required in the horizontal layout. Lemire *et al.* use the 4-way vertical layout (k=4) to SIMDize the bit-unpacking for 32-bit integers on CPUs with `SSE` registers [16]. Also, Habich *et al.* use 8-way and 16-way vertical layouts for AVX2 and AVX512 registers [10]. However, these layouts do not cover all challenges that have been discussed earlier in Table 1: these layouts are tied to a specific SIMD-width, they do not address the problem of sequential data dependencies in LWCs that work on the decoded data (such as DELTA), and do not address the issue of different data type widths in relation to that.

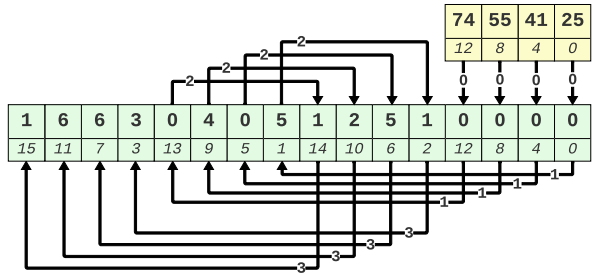
Figure 13 shows that the 4-way layout becomes only slightly faster on AVX2 and AVX512 ISAs. On the other hand, the interleaved layout becomes respectively 2x and 4x faster on AVX2 and AVX512. This confirms that the 4-way layout cannot take advantage of wider registers, while the 1024-bit interleaved layout can.

In addition to the bit-packed layouts that focus on decompression speed, there are other bit-packed layouts that focus more on the filter scan. BitWeaving [18] and ByteSlice [8] are two examples of such layouts. BitWeaving proposes two novel bit-packed data layouts: HBP and VBP. These layouts allow using all the bit-parallelism of a SIMD register during the filter scan. HBP is more focused on supporting efficient lookup operations, while VBP provides a faster filter scan. ByteSlice tries to achieve both fast lookup and fast filter scan by applying all the BitWeaving techniques in the byte-by-byte manner instead of bit-by-bit. However, neither BitWeaving nor BitSlice provides a fast and efficient way to actually decompress data. Polychroniou *et al.* propose a SIMDized bit-unpacking for the VBP layout [24]. However, the reported performance of this layout is roughly 30x slower than our 1024-bit interleaved layout.

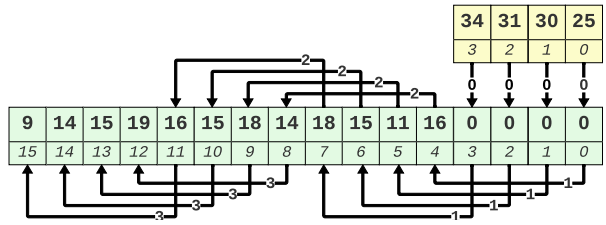
DELTA coding is an LWC that encodes a sequence of integers by replacing each integer with its difference to its preceding integer [19]. DELTA is typically used on top of bit-packing to reduce the number of bits required to represent values. While improving the compression ratio, DELTA decoding becomes a bottleneck in combination with bit-unpacking. Three approaches have been proposed to data-parallelize DELTA decoding: vertical computation [36], horizontal computation [11] [17], and the SIMDized tree computation [36]. Vertical computation is based on the SIMD `SCATTER/GATHER` instructions

Approach	Decompression Cost	Compression Overhead	Shortcoming
Scalar [19]	S	0	Data dependent
Four Cursor [3]	S	$\frac{4 \cdot T}{N}$	Data dependent
Vertical [36]	2	0	Random access
Horizontal [11]	$\log S$	0	Not efficient
Tree [36]	2	0	Random access
D4 [16]	1	$\log S$	Compression ratio
DM [16]	2	$\log \frac{(S+S-1+\cdots+1)}{S}$	Compression ratio
Unified Transposed Layout	1	$\frac{1024}{S}$	-

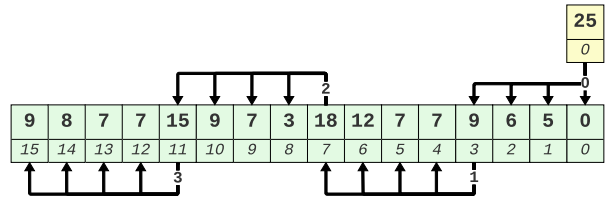
Table 3: Summary of all proposed approaches for SIMD DELTA decoding. Decompression Cost is the number of ADD instructions required to decode S values, while the Compression Overhead is the number of extra bits required.



(a) Unified Transposed Layout. The vector can be bit-packed using 3 bits per value as the maximum delta is 6.



(b) D4 data layout. The maximum delta is now 19. Therefore, 5 bits are required to bit-pack each value.



(c) DM data layout. The maximum delta is now 18. Therefore, 5 bits are required to bit-pack each value.

Figure 14: The Unified Transposed layout needs fewer bits than D4 and DM as it keeps DELTAs between subsequent values.

with non-sequential access pattern. Unfortunately, these instructions are costly and do not make decoding faster [36]. Horizontal computation reduces the complexity of DELTA decoding from $O(n)$ to $log(n)$. This is achieved by using the SIMD SHIFT instructions. However, these instructions do not exist in all ISAs, and it is costly to simulate them. Finally, the tree approach is based on Guy *et al.*'s work [4] and also relies on SCATTER/GATHER instructions [36].

The SIMD implementation of horizontal computation can be considered state-of-the-art [36]. This implementation depends on

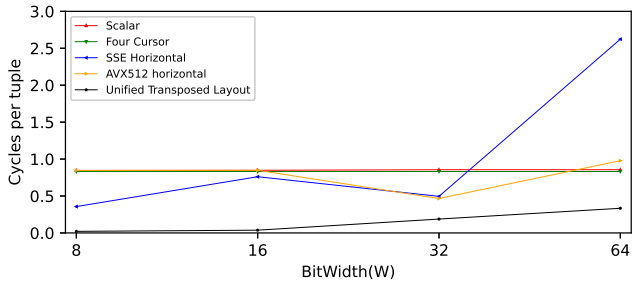


Figure 15: DELTA decoding on the Unified Transposed layout is 3x-40x faster than the alternatives (Ice Lake). Note the AVX512 horizontal computation falls back to scalar for $T = 8$ and $T = 16$ as it requires the _mm512_alignr_epi instruction.

the SHIFT instruction that shifts bits together arbitrarily times to the right. However, this instruction only exists for SSE registers. Zhang *et al.* propose to extend this implementation to AVX-512 by simulating the SHIFT instruction with two SET, and ALIGNR instructions [36]. This implementation needs 12 instructions for every 16 integers. Compared to FastLanes, we can see that this SIMDization does not address all the challenges mentioned earlier. First, data dependency still exists. Second, these implementations are not designed to support all SIMD ISAs.

Rather than SIMDizing the decoding part of the naive DELTA layout, several studies have focused on changing the data layout of DELTA. Lemire *et al.* [16] has proposed two approaches: DM and D4. The key idea behind these two approaches is to keep deltas between adjacent batches of values instead of adjacent values. As shown in Figure 14b, D4 subtracts the values batch-wise, while DM (Figure 14c) subtracts the last value of the previous batch with the next batch. Although D4 provides more data parallelization, the problem here is that the DELTAs are bigger because they are the difference between more distant values. In D4, the differences are 4x bigger, which reduces the compression factor typically by $\log_2(4)$, hence a factor 2. Unfortunately, to support ever wider SIMD registers, ever larger batches are necessary, increasing this overhead.

Another layout proposed to mitigate the issue of data dependency is the four cursors layout [3]. The key idea is to keep more base values, so we can decode more values in parallel without dependencies. This layout was already shown in Figure 4c. Note that although we cannot use SIMD instructions to decode these four values simultaneously, it allows a wide-issue scalar CPU to achieve better ILP by working on four cursors inside one same scalar loop.

Figure 15 shows the performance of the DELTA decoding methods summarized in table 3. The performance of the horizontal methods is inconsistent, as important SIMD instructions are not available for all register- and lane-width combinations. Four-cursor improves Scalar a little. The Unified Transposed layout is by far fastest. It does increase the amount of base values per vector: from 1 to S (the amount of lanes, $1024/T$). The bit-packed vector with deltas takes W*1024, and each base W bits, so the overhead is 1 bit per value. But bases are ascending, so one could DELTA-encode all bases of consecutive vectors in a row-group header. As each vector has T values per lane, and the sum of T W-bit values needs W+log(T) bits, a DELTA-encoded base can be stored in W+log(T)+1 bits, where the +1 is because these bases also need (uncompressed) bases. As 1024 main values need 1024/T bases, DELTA-encoding bases reduces

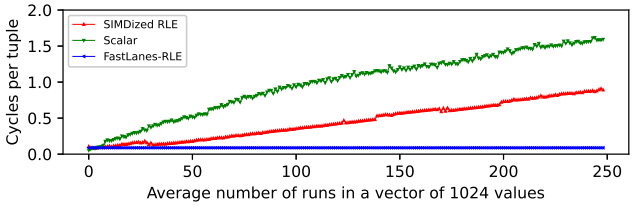


Figure 16: RLE decoding: Scalar, vs SIMDized vs FastLanes-RLE (Ice Lake). FastLanes-RLE is much faster except whith run lengths >333, i.e. at avg 3 runs in a 1024-value vector.

base-overhead from 1 to $(B+\log(T)+1)/T$ bits per value. For example, for the T=64-bit data type, and DELTAs that fit W=7 bits, the extra cost is:((7+log(64)+1)/64)=0.21 bit per value. So that turns W=7 bits per value into 7.21 bits per value (3% overhead).

RLE has been shown to be useful in column-oriented databases [2]. Compared to other LWCs, RLE is fundamentally different: While other LWCs represent the original data as a sequence of small integers, RLE reduces the number of values required to represent the original data. This makes it very challenging to data-parallelize RLE, as we are dealing with a variable number of values. Nonetheless, there were several attempts to SIMDize RLE. The encoding part of RLE has been SIMDized in [15, 21, 31]. For the decoding part of RLE, Damme *et al.* propose a new implementation that could be considered the state-of-the-art [7]. We discussed this scheme when we introduced FastLanes-RLE and call it SIMDized RLE here.

Figure 16 shows that FastLanes-RLE is significantly faster than the other solutions, when runs are longer than 70 (i.e. less than 15 runs in the 1024-value vectors we test on). This is because of two reasons. First, the SIMDized RLE and Scalar suffer from branch miss predictions. This happens in case of storing a new run, as there is a need to take another path to load the new value, and the branch happens more frequently as there are more runs. Second, the SIMDized RLE approach does not profit from the full width of a SIMD register. This is because the next `STORE` instruction may overwrite most of the values stored by the previous `STORE` instruction.

When introducing FastLanes-RLE, we already mentioned its compression ratio is better for runs with an average length ≤12 (in Figure 16, for more than 80 runs in a vector), but starts suffering for longer runs, as its Run Lengths require 1.375 bits per value (W=1 + (1+log(16)+1)/16 for bases, since FastLanes-RLE relies on W=1, T=16 FastLanes-DELTA). However, RLE compression ratio typically does not depend so much on Run Lengths as on Run Values, certainly if these are strings. Also, our future work on cascading encodings (i.e. compressing Run Values, and DELTA-bases) and exception handling schemes, will improve the compression ratio of FastLanes-RLE, by moving to 0-bit DELTA storage with the 1-bits as exceptions, for vectors with long runs.

5 CONCLUSION AND FUTURE WORK

Current database systems only profit to a limited extent from what SIMD could bring [23, 24, 38]. With stalling progress in CPU frequency and core counts, this is still an opportunity for performance gains. In our vision, one needs to start by redesigning the basis – data storage – to seize this opportunity. This is why FastLanes proposes a new data layout, that creates opportunities for independent work on data-parallel hardware. Besides SIMD, we remark that other popular data-parallel hardware includes GPUs and TPUs and that we are in an age of further hardware innovation. The gist of FastLanes is that this age needs a data format that takes away sequential decoding dependencies and that is why its key idea is to reorder tuples in the special "04261357" 8x16 tiling order.

FastLanes can express all common LWC decoding methods in simple operations on a virtual (and future-proof) 1024-bits register that can efficiently map to existing SIMD instruction sets, as shown by our experiments on Intel, AMD, Apple and AWS hardware.

Rather than looking at value decoding in isolation, we look at it from a database systems context, where decompression is part of a pipeline that should be in balance with hardware resource limits, and where a column is not decoded fully in isolation, but incrementally (vector-at-a-time), as the source of a query pipeline, that processes the data further, and where the scan decodes multiple different columns. And, where decoding infrastructure is part of a (vectorized) software subsystem [13], where code portability in an ever more heterogeneous hardware environment is of paramount importance, to limit development effort and technical debt.

FastLanes also has a scalar code-path, and the data-paralellism on compact data-types that it exposes, even accelerates scalar decoding in comparison with naive bit-packed sequentially stored data. A key result is that modern compilers can completely auto-vectorize this scalar code-path, with no performance penalty compared to explicit SIMD intrinsics. This makes FastLanes very portable.

The performance benefits of FastLanes start by providing **much** faster decompression: our bit-unpacking followed by FOR and DELTA decompression improve over naive sequential bit-packed layouts by often an order of magnitude (or more). We showed that RAM-resident queries can get even faster on FastLanes-compressed data, when compared with direct in-memory array scans.

Future Work. Our proposed kernels, such as FastLanes-RLE are not targeting full/eager decompression, but rather partial decompression into *compressed vector* representations. Such vector representations, that represent vectors of data in tight arrays that fit in a lane-width that is much smaller than the fully decompressed value, unlock opportunities for relational operators higher up in the pipeline to exploit compressed execution [2, 6, 20, 25, 33, 34].

Research could establish whether the data-parallelism that FastLanes creates makes it also suitable to efficiently scan and process data on widely-parallel hardware such as TPUs and GPUs [28].

In FastLanes we aim not only to improve the speed of LWC decoding, but also the compression ratio. We are researching the idea of *cascading* LWCs [26], where compression methods are stacked on top of each other, and combined with various exception handling schemes; with the ultimate goal of making general-purpose compression methods such as zstd, Snappy and (even) LZ4 less necessary in big data formats; as their decoding speeds are orders of magnitude slower than FastLanes, and holding back performance.

We leave an evaluation in a complete system on end-to-end benchmarks for future work. We intend to integrate FastLanes in a complete open source future-proof big data file format. Cascading compression implies that each logical column chunk gets stored in potentially multiple recursively compressed physical sub-column-chunks, and this involves making and evaluating many design decisions in row-group, data-chunk and meta-data organization.

REFERENCES

[1] [n.d.]. Apache Parquet. http://parquet.apache.org/.

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 671–682.

[3] A Afroozeh. 2020. Towards a New File Format for Big Data: SIMD-Friendly Composable Compression. https://homepages.cwi.nl/~boncz/msc/2020-AzimAfroozeh.pdf

[4] Guy E. Blelloch. 2004. Prefix sums and their applications. (5 2004). https://doi.org/10.1184/R1/6608579.v1

[5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[6] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12(12) (2019), 2022–2034.

[7] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*.

[8] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. 2015. ByteSlice: Pushing the Envelop of Main Memory Data Processing with a New Storage Layout. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 31–46.

[9] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida, USA, February 23-27, 1998*, Susan Darling Urban and Elisa Bertino (Eds.). IEEE Computer Society, 370–379.

[10] Dirk Habich, Patrick Damme, Annett Ungethüm, and Wolfgang Lehner. 2018. Make Larger Vector Register Sizes New Challenges? Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages.

[11] W. Daniel Hillis and Guy L. Steele. 1986. Data Parallel Algorithms. *Commun. ACM* 29, 12 (dec 1986), 1170–1183.

[12] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (2018), 2209–2222.

[13] 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* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 311–326.

[14] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. *The VLDB Journal* 29, 2 (01 May 2020), 757–774.

[15] Florian Lemaitre, Arthur Hennequin, and Lionel Lacassagne. 2020. How to Speed Connected Component Labeling up with SIMD RLE Algorithms. In *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing* (San Diego, CA, USA) *(WPMVP'20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages.

[16] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45 (01 2015).

[17] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD Compression and the Intersection of Sorted Integers. *Softw. Pract. Exper.* 46, 6 (jun 2016), 723–749.

[18] Yinan Li and Jignesh Patel. 2013. BitWeaving: Fast scans for main memory data processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 289–300.

[19] Wee Keong Ng and Chinya V. Ravishankar. 1997. Block-Oriented Compression Techniques for Large Statistical Databases. *IEEE Trans. on Knowl. and Data Eng.* 9, 2 (March 1997), 314–328.

[20] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384.

[21] Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, and Wolfgang Lehner. 2018. Beyond Straightforward Vectorization of Lightweight Data Compression Algorithms for Larger Vector Sizes. In *Grundlagen von Datenbanken*.

[22] Jeff Plaisance, Nathan Kurz, and Daniel Lemire. 2015. Vectorized VByte Decoding. *ArXiv* (2015).

[23] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *ACM SIGMOD*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1493–1508.

[24] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (Melbourne, VIC, Australia) *(DaMoN'15)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages.

[25] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://www.duckdb.org

[26] Vijayshankar Raman and Garret Swart. 2006. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 858–869.

[27] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast integer compression using SIMD instructions. 34–40.

[28] 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.

[29] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi. 2011. SIMD-Based Decoding of Posting Lists. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 317–326.

[30] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2018. The ARM Scalable Vector Extension. *CoRR* abs/1803.06185 (2018).

[31] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2018. Conflict Detection-Based Run-Length Encoding - AVX-512 CD Instruction Set in Action. 96–101.

[32] Annett Ungethüm, Johannes Pietrzyk, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner, and Erich Focht. 2020. Hardware-Oblivious SIMD Parallelism for In-Memory Column-Stores. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[33] Richard Michael Grantham Wesley and Pawel Terlecki. 2014. Leveraging Compression in the Tableau Data Engine. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 563–573.

[34] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (sep 2000), 55–67.

[35] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan Using on-Chip Vector Processing Units. *Proc. VLDB Endow.* 2, 1 (aug 2009), 385–394.

[36] Wangda Zhang, Yanbin Wang, and Kenneth Ross. 2020. Parallel Prefix Sum with SIMD. (09 2020).

[37] Wayne Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-yun Nie, Hongfei Yan, and Ji-Rong Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Transactions on Information Systems* 33 (02 2015).

[38] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 145–156.

[39] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 59.