Short Paper

Towards A General SIMD Concurrent Approach to Accelerating Integer Compression Algorithms

Juliana Hildebrandt
TU Dresden
Dresden, Germany
juliana.hildebrandt@tu-dresden.de

Dirk Habich
TU Dresden
Dresden, Germany
dirk.habich@tu-dresden.de

Wolfgang Lehner
TU Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

Integer compression algorithms play an important role in column-oriented data systems. Previous research has shown that the vectorized implementation of these algorithms based on the Single Instruction Multiple Data (SIMD) parallel paradigm can multiply the compression as well as decompression speeds. While a scalar compression algorithm usually compresses a block of N consecutive integers, the state-of-the-art SIMD implementation scales the block size to $k * N$ with k as the number of elements which could be simultaneously processed in a SIMD register. However, this means that as the SIMD register size increases, the block of integer values for compression also grows, which can have a negative effect on the compression ratio. In this paper, we analyze this effect and present an idea for a novel general approach for the SIMD implementation of integer compression algorithms to overcome that effect. Our novel idea is to concurrently compress k different blocks of size N within SIMD registers. To show the applicability of our idea, we present initial evaluation results for a heavily used compression algorithm and show that our approach can lead to more responsible usage of main memory resources.

1 INTRODUCTION

Column-store database systems are state-of-the-art for analytical application scenarios [3, 11]. These systems have in common that all values of every column are encoded as a sequence of integer values and, thus, query processing is done on these integer sequences [1, 6]. The necessary memory space for storing these integer sequences can be reduced with the help of some additional lightweight computations for integer compression. Moreover, compressed integer values offer advantages for data processing such as increasing the effective bandwidth to reduce the memory wall or a better utilization of the cache hierarchy. As shown in [4, 5, 12, 15], there is a large number of lightweight integer compression schemes available. Nevertheless, compression as well as decompression leads to additional computational effort, which is typically kept low by means of vectorization.

Vectorization based on the *Single Instruction Multiple Data (SIMD)* parallel paradigm is a state-of-the-art technique, because all mainstream CPUs offer powerful SIMD extensions nowadays [10]. The main objective of SIMD is to increase the single-thread performance by executing an identical operation on multiple data elements in a vector or SIMD register simultaneously (data parallelism) [10]. In recent years, the efficient SIMD-based implementation of these integer compression algorithms has attracted a lot of attention [4, 5, 15], since it further reduces the computational effort. As described in [15], the general state-of-the-art SIMD-based implementation can be described as follows: *While a scalar compression algorithm would compress a block of N consecutive integers, the state-of-the-art SIMD approach scales this block size to $k * N$ with with k as the number of integers which can be simultaneously processed within a SIMD register.*

A current hardware trend is to increase the size of the SIMD registers. On the one hand, Intel's latest SIMD extension AVX-512 uses 512-bit SIMD registers, while previous SIMD extensions of Intel operate on 128-bit (SSE) or 256-bit (AVX2). On the other hand, the ARM NEON extension (available in ARMv7-A and ARMv8-A) was limited to a SIMD register size of 128-bit, the recently announced Scalable Vector Extension (SVE) (available in ARMv8-A AArch64) aims at supporting much wider SIMD registers from 128 to 2,048-bits, in 128-bit increments [14]. The wider the SIMD registers, the more data elements can be stored and processed simultaneously, which promises significant speedups. However, this means that as the SIMD register size increases, the block of integer values for the state-of-the-art SIMD implementation for compression also grows, which can have a negative effect on the compression ratio.

Our Contribution and Outline. In this paper, we analyze this effect and show that the compressed output of a state-of-the-art SIMD implementation could be many times larger than the result of a scalar implementation of the same compression algorithm. Thus, this is not conducive to reducing the memory footprint. To overcome that, we present an idea for an alternative generalized approach for the SIMD-based implementation by concurrently compressing k different blocks of size N within SIMD registers. To present our approach, the remainder of the paper is structured as follows: In Section 2, we briefly summarize the state-of-the-art in the domain of integer compression algorithms including the SIMD-based implementation. Moreover, we theoretically analyze the compression ratios of the scalar and SIMD-based implementations of a heavily used and representative compression algorithm. Then, we present our alternative generalized approach for the SIMD-based implementation in Section 3. In particular, we will (i) discuss different implementation choices for our running representative compression algorithm, (ii) present some initial evaluation results, and (iii) discuss our ongoing research work. We close the paper with related work in Section 4 and a short summary in Section 5.

2 STATE-OF-THE-ART

The objective of every lossless, lightweight integer compression algorithm is to represent a finite sequence of integer values with as few bits as possible [4, 5, 12, 15]. Over the past few decades, a large corpus of different scalar algorithms has evolved [4, 5, 12, 15]. The algorithms have in common that the compressed representation for single or blocks of integer values often consists of control patterns and data snips [15]. Data snips represent the compressed integers in binary format, while control patterns store the auxiliary information to interpret the data snips.

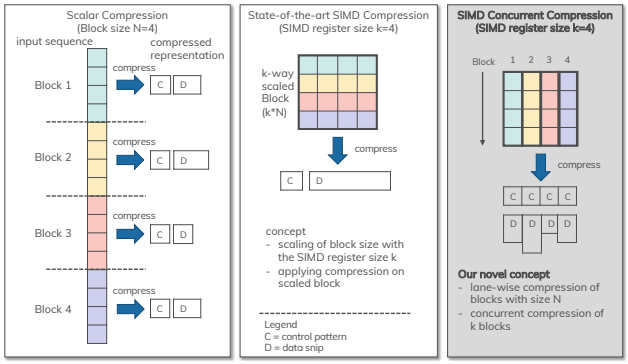


Figure 1: Overview of compression processing concepts.

A well-known and representative integer compression algorithm is *BitPacking (BP)* belonging to the class of null suppression algorithms by omitting leading zero bits [12]. We use *BP* as running example throughout the paper. The scalar version of *BP* for 64-bit integer values is called *BP64* and works as follows: A finite input sequence of integer values is subdivided into blocks of 64 integers each. For each block, the minimal number of bits required for the largest element is determined. Then, all 64 integers in each block are stored in data snip with with the respective number of bits for each value. The used bit width is stored in a single byte as control pattern. Other scalar compression algorithms operate in a similar way and Figure 1 gives a schematic overview about this procedure using a block size of four.

As described in [15], the state-of-the-art generalized SIMD approach is characterized by the fact, that (i) the block size is scaled by the SIMD register size k – k is the SIMD register size in number of integer values — and (ii) the application of the compression on this larger block size. For example, *SIMD-BP256* is the SIMD-based implementation of our running example algorithm *BP64* for SIMD register sizes of 256-bits. In those SIMD registers, we are able to store and process 4 64-bit integer values at once, so that the block size is scaled by $k = 4$. Based on that, the k-way scaled SIMD block contains 256 integer values and for these 256 elements, the minimal number of bits required for the largest element is determined. Then, all 256 integers in each block are stored in a data snip with that many bits for each value and the used bit width is stored as common control pattern.

Advantages of this generalized k-way scaling concept for SIMD are: (i) in many cases, scalar algorithms can be easily ported to SIMD-based implementations, (ii) the generalized approach supports different SIMD register sizes, and (iii) the SIMD-based implementations are faster than the scalar versions [4, 5, 12, 15]. However, a main drawback of this k-way scaling is that the compression factor

$$cf(k) = \frac{|\text{k-way compressed data}|}{|\text{uncompressed data}|} \tag{1}$$

mostly increases with an increasing SIMD register and block size. On the one hand, fewer control patterns need to be stored due to larger block sizes. On the other hand, a number of larger integer values may be compressed with a larger bit width. To precisely analyze this effect, we use our running example algorithm and derive the expected compression factor for different integer bit width distributions.

Our analysis framework works as follows: The scalar algorithm *BP64* encodes blocks of 64 64-bit values with the least possible common bit width and the bit width as control pattern itself with 64 bits. The SIMD-based implementations with SIMD register size k encode $64 \cdot k$ values with the same approach. Given is a data distribution for 64-bit integer values characterized by the probability for the bit widths $0 \leq b \leq 64 : p(b)$ and a SIMD register size k. Now, we can distinguish 65 cases corresponding to blocks of $64 \cdot k$ values which are encoded with bit width $0 \leq b \leq 64$. Each of these cases (i) occurs with a probability $p'(b,k)$, which depends on the given data distribution and the SIMD register size k , and (ii) is characterized by a block compression factor $cf'(b,k)$. The expected compression factor for a k-way SIMD-based implementation of *BP* can be calculated by

$$cf(k) = \sum_{b=0}^{64} p'(b,k) \cdot cf'(b,k). \tag{2}$$

The block compression factor $cf'(b,k)$ is given by

$$cf'(b,k) = \frac{|k\text{-way compressed block}|}{|\text{uncompressed block}|} = \frac{1 + b \cdot k}{64 \cdot k} \tag{3}$$

and the block probability can be derived by the following consideration. The probability of the occurrence of a block of size $64 \cdot k$ containing only zero values is $p'(0,k) = p(0)^{64 \cdot k}$. The probability of a block encoded with one of the bit widths $0 \leq b$ is $\left(\sum_{bw=0}^{b} p(bw)\right)^{64 \cdot k}$. The probability for the occurrence of a block encoded with bit width b is the difference of the above probability and all probabilities for the occurrences of blocks with a smaller bit width than b:

$$p'(b,k) = \left(\sum_{bw=0}^{b} p(bw)\right)^{64 \cdot k} - \sum_{bw=0}^{b-1} p'(bw,k). \tag{4}$$

For the compressed size ratio between a k-way SIMD-based implementation and the scalar implementation of *BP*, we calculate $\frac{cf(k)}{cf(1)}$ with $cf(1)$ corresponding to the scalar compression factor (scaling factor $k = 1$).

In the following, we apply these formulas on two different data distributions where most integer values are characterized by a bit width of 2, but we also have a probability x for integer values with a larger bit width. These larger integer values are denoted as outliers. While in the first case the outliers have a bit width of 3, the outliers in the second case have a bit width of 60. Figure 2 depicts the compressed size ratio $\frac{cf(k)}{cf(1)}, k = \{2, 4, 8, 16, 32\}$ for the SIMD-based implementations for both cases and different outlier probabilities. As we can observe, all lines are below 1 for case one with the outlier bit width of 3. That means, each SIMD-based implementation (using different k-way scalings) has a lower compression factor than the scalar algorithm. The reason is the lower number of control patterns for larger blocks and the more or less homogeneous bit widths for all integer values. Moreover, the value k for the best SIMD-based implementation yielding the best compression ratio depends on the outlier probability.

In contrast to that, for the second case with the outlier bit width of 60 and lower outlier probabilities, we see that all lines are above 1. This means, the compression ratio of the scalar variant is much better than of the SIMD-based implementations. For example, a compressed representation of 8-way SIMD-based implementation of *BP* – SIMD register size 512-bit with 64-bit integer values – is 4 times larger than for the scalar variant. Those disturbing effects happen and destroy the advantages of the SIMD-based integer compression, especially since a small number of outliers has such large effects.

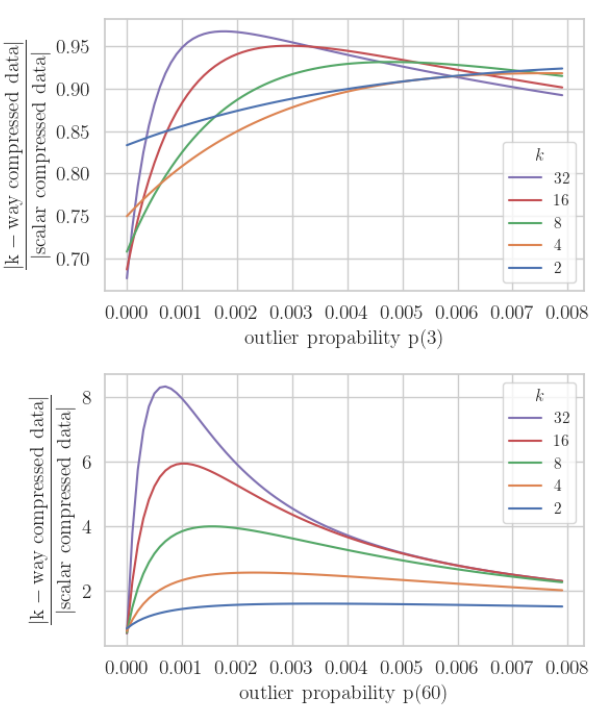


Figure 2: Ratio of SIMD and scalar compressed data size.

3 BLOCK CONCURRENT COMPRESSION

From the previous section and our experimental evaluations in [4, 5, 7], we conclude that the state-of-the-art SIMD-based approach optimizes the compression algorithms from a performance perspective. However, this approach has shortcomings from a memory footprint point of view. To overcome that, we propose an alternative generalized SIMD approach and explain the application to our running example algorithm *BP*. Afterwards, we present some initial evaluation results and discuss our further research activities in that direction.

General Idea. Instead of scaling the block size by the SIMD register size k, we suggest a block concurrent compression concept as generalization as depicted in Figure 1. In this block concurrent concept, each SIMD register place – also called SIMD lane – compresses its own data block. Thus, the number of available SIMD lanes determines the number of blocks which will be compressed simultaneously. The advantages are (i) the same block size of the scalar compression algorithms is maintained and (ii) the control patterns as well as data snips are calculated lane-wise. That means, we apply the scalar compression algorithm on each SIMD lane on different data blocks concurrently. A shortcoming could be that integer values from different memory regions must be loaded into vector registers.

Application to BP. Figure 3 illustrates the realization of *BP* with our block concurrent concept. Here, we assume integer values of size 64-bit, which will be compressed with the scalar variant *BP64* as introduced above. The assumed SIMD register size k is 8, so that eight different data blocks of 64 values are compressed simultaneously. That means, we are processing 512 integer values in total of the finite input sequence of integer values and compute eight control patterns and data snips as compressed output. The *BP* compression is done in two phases, thereby each phase iterates over all 512 integer values. In the first phase, the bit width of the largest integer value within each

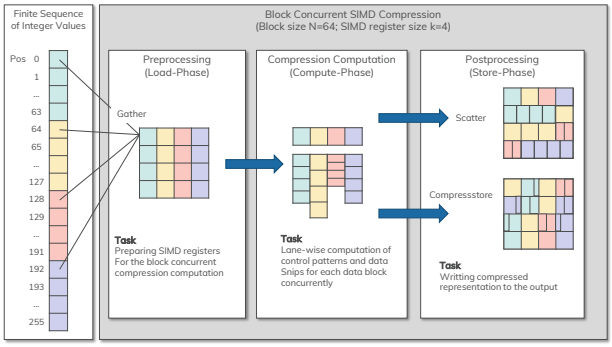


Figure 3: Block concurrent BP compression.

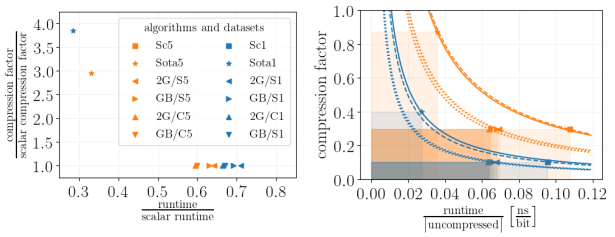
block is determined, while the second phase uses the determined bit widths to shorten the values accordingly and to finally write out the compressed representation.

As shown in Figure 3, we distinguish (i) a preprocessing step to load the data from the input into the vector registers, (ii) a computation step to shorten the values, and (iii) a postprocessing step to write the data into the output area. These steps are executed in each phase and each phase executes 64 iterations for 64 values per block. That means, for the *nth* iteration, we require the integer value of the *nth* position of each considered data block in the SIMD register. Assuming that the input sequence contains correct ordered data (*horizontal data layout*), we simply could use a `SIMD-gather` operation to load the corresponding values of the different data blocks into the SIMD register. The `SIMD-gather` seems an expensive operation, because it can be used for random memory access. However, in our case, we realize a strided access with a distance of 64.

In the computation step, we apply the appropriate SIMD functions for each phase. In the first phase, we apply the SIMD functions to compute the number of leading zeros for the largest value per lane (per block). Based on the number of leading zeros, we compute the minimal number of bits for the compression. This bit widths are used in the second phase to concatenate the shortened data values while using an appropriate SIMD-bitshifting operation, which can be applied for each lane individually. Because in the single lanes, the data is concatenated with a different bit width, the lanes are filled at different loop passes. For example, a lane is full after 2 iterations for a bit width of 30 (assuming 64 bit integer values), but for a bit width of 2, we need 32 iterations to fill a lane. In any case, if one of the lanes is full, it has to be written to the output (postprocessing step). Here, we see two alternatives. The first alternative is to use a `SIMD-compressstore`-operation to consecutively write out lanes as soon as they are full. In this case, data snips of the different blocks are intertwined. The second alternative is to use a `SIMD-scatter`-operation. Since the bit width for each block is determined at first, the bit widths can also be used to calculate the position for each full lane in the output. In this case, we are able to organize the data snips for each of them in a consecutive manner.

Since we read the input data twice, we must also perform the `SIMD-gather` operation twice. To avoid this, we can optimize this by using a buffer with a size of 512 ($8 \cdot 64$) values. In the first phase, this buffer is filled with the gathered input data, while in the second phase, we only to consecutively load the data from the buffer. In this case, we save one `SIMD-gather` operation.

Evaluation. To evaluate whether and when our general idea is suitable, we implemented our running compression algorithm



(a) Space vs. Time. (b) Space Time Products (STP).

Figure 4: Compression and runtime comparison for the following implementation variants: scalar (Sc), state-of-the-art SIMD (Sota) and our proposed block concurrent implementations (2G/S, GB/S, 2G/C, GB/C) for two outlier probabilities ($p = 0.001$ and $p = 0.005$ denoted by the suffixes 1 and 5 in the labels).

example *BP* in its scalar form and with the state-of-the-art SIMD[1] as well as our novel block concurrent approach using Intel's latest SIMD extension AVX-512. For the block concurrent approach ($k = 8$), we implemented the following variants as described above: (i) 2x gather with scatter (denoted as 2G/S), (ii) gather with buffer and scatter (GB/S), (iii) 2x gather with compressstore (2G/C), and (iv) gather with buffer and compressstore (GB/C). The state-of-the-art SIMD implementation only uses `SIMD-load` and `SIMD-store` operations with a block scaling factor $k = 8$. We compiled our source code using `g++` (version 9.3.0) with the optimization flags `-O3 -fno-tree-vectorize -mavx512f -mavx512cd`. We ran our evaluation on an Intel Intel(R) Xeon Phi(TM) CPU 7250 with 204GB DDR4 main memory (the available high-bandwidth memory called MCDRAM was neither used explicitly nor configured as L3 cache). All experiments are executed single-threaded, happened entirely in-memory, were repeated 10 times, and we averaged the results.

For our experiments, we only use the setting of the analysis from Section 2, where the compression ratio of the state-of-the-art SIMD approach was poor, so that the compression ratio of the block concurrent approach is the same as for the scalar implementation as thus much better compared to the state-of-the-art SIMD approach. For that, we generated synthetic data, where most integer values are characterized by a bit width of 2 and we varied the probability for integer values with a larger bit with of 60. The resulting performances in *runtimes normalized by the scalar runtimes and compression rates normalized by the scalar compression rates* are depicted in Figure 4a for two outlier probabilities $p = 0.001$ and $p = 0.005$. As we can see, the block concurrent approach achieves lower normalized-to-scalar runtime performances than the state-of-the-art implementation (0.66 vs. 0.28 for $p = 0.001$ and 0.59 vs. 0.33 for $p = 0.005$), but better runtime performances than the scalar implementation on the one hand. On the other hand, the normalized-to-scalar compression factor of the block concurrent approach is much better than for the state-of-the-art SIMD case (3.8 vs. 1.0 for $p = 0.001$ and 2.9 vs. 1.0 for $p = 0.005$). To compare the measurements in time and space in equal way, our goal is to maximize the uncompressed data size per compressed data size and time respectively for a given uncompressed data size to minimize the product of compressed data size and time. Because of the square influence of the uncompressed data size, we define the space time product (stp) as

$$stp = \frac{|compressed| \cdot runtime}{|ucompressed|^2} = \frac{cf \cdot runtime}{|uncompressed|}, \quad (5)$$

and thus, determine the product of space in bits and time in nanoseconds that is needed for each uncompressed bit to perform the compression. The less the stp of a compression process, the more economically the limited memory is used. Figure 4b shows the stp of each implementation as an area as well as the tuples of all compression factors and $\frac{runtime}{|uncompressed|}$ fractions leading of the same stp as lines. The block concurrent implementations are much better than the state-of-the-art SIMD implementation for both outlier probabilities (0.0066 ns/bit vs. 0.0108 ns/bit for $p = 0.001$ and 0.0190 ns/bit vs. 0.030 ns/bit for $p = 0.005$).

Ongiong Research Activities. Due to the promising results, we want to intensify our work in this area and refine our approach by looking at different exiting integer compression algorithms. From our point of view, the preprocessing and postprocessing components are the biggest challenges, since often the block size are varied depending on the data. In addition, a comprehensive comparison with the state-of-the-art SIMD approach must be carried out to work out the advantages and disadvantages.

4 RELATED WORK

A comprehensive overview of the field of lossless lightweight integer compression algorithms and SIMD implementations is given by the following papers [4, 5, 12, 15]. In addition to that, we presented a meta-model to specify integer compression algorithms in a descriptive and abstract way with the ability to derive executable code from that description [8, 9]. An integration of our presented generalized SIMD approach into the transformation to the executable code is in the focus of our ongoing research activities. Moreover, the selection of the best-fitting integer compression variant is a research field with a very dynamic development [2, 5]. With our alternative generalized SIMD approach, we extend the variety of variants increasing the importance of the selection. From a SIMD execution point of view, our presented generalized SIMD approach is in line with the idea of sharing vector registers for concurrently running queries as described in [13]. Nevertheless, the application as well as the specific challenges differ. However, both approaches show that an alternative use of SIMD execution can be profitably employed.

5 CONCLUSION

Integer compression algorithms play an important role to reduce the memory footprint and to speedup query processing in column-stores. While a scalar compression algorithm usually compresses a block of N consecutive integers, the state-of-the-art SIMD implementation usually scales the block size to $k \cdot N$ with k as the number of elements that could be simultaneously processed in a SIMD register. However, this means that as the SIMD register size increases, the block of integer values for compression also grows, which can have a negative effect on the compression ratio. In this paper, we analyzed this effect and showed that the compressed output could be many times larger than the result of a scalar implementation. To overcome that, we presented an approach towards a novel general approach for the SIMD implementation by concurrently compressing k different blocks of size N within SIMD registers of size k. Moreover, we showed some initial evaluation results for a heavily used compression algorithm. Our block concurrent implementation can lead to more responsible usage of main memory resources.

[1]Both implementations for 64-bit integers are inspired by existing implementations of Daniel Lemire for 32-bit integers published on Github: https://github.com/lemire/LittleIntPacker/blob/master/src/bitpacking32.c, https://github.com/lemire/simdcomp/blob/master/src/avx512bitpacking.c

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. 671–682.

[2] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *EDBT*. 674–677.

[3] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.

[4] Patrick Damme et al. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. 72–83.

[5] Patrick Damme et al. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46.

[6] Patrick Damme et al. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 11 (2020), 2396–2410.

[7] Dirk Habich et al. 2018. Make Larger Vector Register Sizes New Challenges?: Lessons Learned from the Area of Vectorized Lightweight Compression Algorithms. In *DBTest*. 8:1–8:6.

[8] Juliana Hildebrandt et al. 2017. Metamodeling Lightweight Data Compression Algorithms and its Application Scenarios. In *ER Forum and Demo Track*. 128–141.

[9] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2016. Model Kit for Lightweight Data Compression Algorithms. In *EDBT*. 692–693.

[10] Christopher J. Hughes. 2015. *Single-Instruction Multiple-Data Execution*. Morgan & Claypool Publishers.

[11] Marcel Kornacker et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.

[12] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* 45, 1 (2015), 1–29.

[13] Johannes Pietrzyk, Dirk Habich, and Wolfgang Lehner. 2020. To share or not to share vector registers?. In *DaMoN*. 12:1–12:10.

[14] Nigel Stephens et al. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017), 26–39.

[15] Wayne Xin Zhao et al. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Trans. Inf. Syst.* 33, 3 (2015), 15:1–15:28.