Beyond Straightforward Vectorization of Lightweight Data Compression Algorithms for Larger Vector Sizes

Johannes Pietrzyk, Annett Ungethüm, Dirk Habich, Wolfgang Lehner
Technische Universität Dresden
01062 Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Data as well as hardware characteristics are two key aspects for efficient data management. This holds in particular for the field of in-memory data processing. Aside from increasing main memory capacities, efficient in-memory processing benefits from novel processing concepts based on lightweight compressed data. Thus, an active research field deals with the adaptation of new hardware features such as vectorization using SIMD instructions to speedup lightweight data compression algorithms. Most of the vectorized implementations have been proposed for 128-bit vector registers. A straightforward transformation to wider vector sizes is possible. However, this straightforward way does not exploit the capabilities of newer SIMD extensions to the maximum extent as we will show in this paper. On the one hand, we present a novel implementation concept for run-length encoding using conflict-detection operations which have been introduced in Intel's AVX-512 SIMD extension. On the other hand, we investigate different data layouts for vectorization and their impact on wider vector sizes.

1. INTRODUCTION

The continuous growth of data volumes is still a major challenge for efficient data processing. This applies not only to database systems [6, 16], but also to other areas, such as information retrieval [3, 20] or machine learning [8]. With growing capacities of main memory, efficient analytical in-memory data processing becomes viable [6, 16, 13]. However, the gap between CPUs computing power and main memory bandwidth continuously increases being now the main bottleneck [6]. To overcome this issue, the mentioned application domains have a common approach: (i) encode values of each data attribute as a sequence of integers using some kind of dictionary encoding [1, 5] and (ii) apply lightweight lossless data compression to each sequence of integers. Besides reducing the amount of data, operations can be directly performed on compressed data [8, 1, 11].

For the lightweight lossless compression of a sequence of integers, a large corpus of algorithms has been developed [1, 3, 20, 27, 2, 10, 14, 18, 19]. In contrast to heavyweight algorithms, like arithmetic coding [23], Huffman [12], or Lempel Ziv [26], lightweight algorithms achieve comparable or even better compression rates [1, 3, 20, 27, 2, 10, 14, 18, 19]. Moreover, the computational effort for (de)compression is lower than for heavyweight algorithms. To achieve these unique properties, each lightweight compression *algorithm* employs one or more basic compression *techniques*. There are currently five basic lightweight techniques known: frame-of-reference (FOR) [10, 27], delta coding (DELTA) [14, 18], dictionary compression (DICT) [1, 27], run-length encoding (RLE) [1, 18], and null suppression (NS) [1, 18]. These five techniques address different sub-goals. While FOR, DELTA, and DICT consider the mapping to smaller values, the goal of RLE is to reduce the number of values on the logical level, and NS addresses the *physical* level of bits or bytes to reduce the number of bits per value. This explains why lightweight data compression algorithms are always composed of one or more of these techniques.

In recent years, the efficient *vectorized* implementation of these algorithms using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention [20, 14, 24], since it further reduces the computational effort. Generally, SIMD extensions such as Intel's SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) have been available in modern processors for several years. SIMD instructions apply one operation to multiple elements of so-called *vector registers* at once. The available operations include parallel arithmetic, logical, and shift operations as well as permutations. Most of the developed vectorized implementations of lightweight data compression algorithms have been developed for a fixed vector width of 128 bits (corresponding to Intel's SIMD extension SSE). However, hardware vendors have introduced new SIMD instruction set extensions operating on wider vector registers in recent years. For instance, Intel's Advanced Vector Extensions 2 (AVX2) operates on 256-bit vector registers[1] and Intel's AVX-512 uses even 512-bit vector registers. The wider the vector registers, the more data elements can be stored and processed in one vector. For example, while an SSE 128-bit vector register can store four uncompressed 32-bit data elements, an AVX2 256-bit vector can store eight (*2x*) and an AVX-512 512-bit vector can store 16 (*4x*) of such data elements. Consequently, the SIMD instructions on these wider vector registers can also process *2x* respectively *4x* the number of data elements *in one instruction*, which promises significant speed ups.

To obtain implementations of lightweight compression algorithms for wider vector sizes (AVX2 and AVX-512), the 128-bit implementation can be used as foundation. In a straightforward transformation, the 128-bit SIMD operations can be substituted by the corresponding operations for 256 or 512-bit vectors. This is possi-

30^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.

[1]Note that 256-bit vector registers had already been introduced with Intel's AVX. However, most instructions relevant to lightweight data compression were only introduced with AVX2.

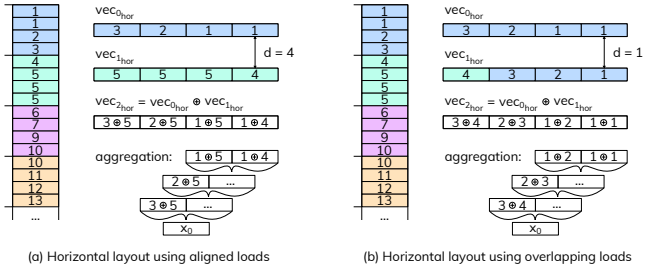

Figure 1: Example for loading and processing data using 128-bit vector registers alongside a horizontal layout.

ble in almost all cases, since many instructions offered by SSE are also offered by AVX2 and AVX-512 *on wider vectors*. However, this straightforward transformation does not exploit the capabilities of newer SIMD extensions to the maximum extent. To show that, we make the following contributions in this paper:

1. We propose a novel vectorization approach for run-length encoding using new instructions which are available in AVX-512 in Section 2. Additionally, we compare our novel approach with the state-of-the-art vectorization and present the benefits of our novel approach.
2. We investigate different data layouts for vectorization, i.e. the horizontal and vertical layout, and their impact on different data sizes in Section 3.

Finally, we review related work in Section 5. Then, we conclude the paper by summarizing our lesson learned and present future work in Section 4 and 6.

2. VECTOR PROCESSING AND NOVEL INSTRUCTIONS IN AVX-512

To use vector instructions for a sequence of data elements, data needs to be transferred from memory into vector registers. This can be done using an aligned load of consecutive data as illustrated in Figure 1(a). Afterwards the registers can be processed using a specific vector instruction. Every element within the first operand vector register is processed with the corresponding element from the second operand vector register. Thus, the nth element from a sequence of data is processed with the $(n + d)$th element, while the distance d equals the number of values that fit into the used vector register. This approach is not well suitable for algorithms which need to process given data with respect to the surrounding data elements. Nevertheless, most vectorized lightweight compression algorithms are based on this *horizontal data layout* [7].

A way to reduce the processable distance d is to use a combination of an aligned load starting with the nth element followed by an unaligned load starting with the $(n + 1)$th element (see Figure 1(b)) [7, 21]. While this method allows to process consecutive data, it needs fairly expensive unaligned loads and doubles the amount of load operations per element. To avoid that, a redesign by considering new vector instructions could be beneficial.

2.1 Novel Instructions

Intel's latest version of their vectorization extension is AVX-512. In addition to an increased vector width of 512-bit (16 x 32-bit), AVX-512 also offers a variety of new instructions. One of the new instruction feature sets is called *Conflict Detection* (AVX-512 CD) which allows the vectorization of loops with possible address conflicts. Some key features of AVX-512 CD are (i) the generation of conflict free subsets, i.e. subsets which contain no equal elements, and (ii) the count of leading zeros of the elements in a vector.

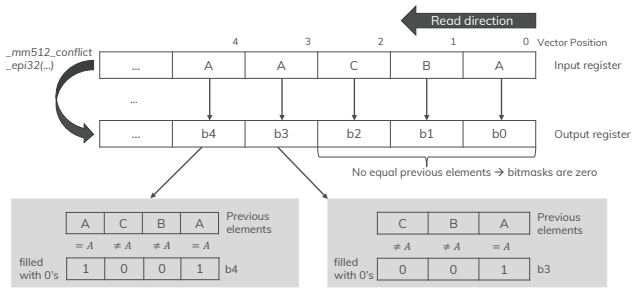

Figure 2: Example for the *_mm512_conflict_epi32* intrinsic.

For example, the intrinsic `_mm512_conflict_epi32` creates a vector register containing a conflict free subset of a given source register. An example for this is shown in Figure 2. In other words and as depicted, this intrinsic transforms a vector register with 16 32-bit elements (illustrated by A, B and C) in a new vector register with 16 bitmasks (each represented by 32-bit values). Each bitmask encodes the positions of equal previous elements in the vector. The bitmasks for the first three elements A, B, and C are zero in our example, because there are no equal previous elements. The A element at the third position in the input register is in conflict (equal to) with the element at position 0 in the input register. Thus, the least significant bit of the corresponding bitmask is set to 1, the rest of the bitmask is filled with zeros. The element A at position 4 is in conflict with the previous elements at positions 3 and 0 (equal previous elements). Therefore, the corresponding bits in the bitmask are set to 1, all other bits are zero. Another CD-feature is the intrinsic `_mm512_lzcnt_epi32`, which counts leading zeros. Given a vector of 16 values, this intrinsic counts the number of leading zeros for all values at once and writes the results in a vector register with 16 values.

To show the impact of these novel instructions, we redesigned Run-Length Encoding (RLE), a well-known lightweight compression technique [1, 18], to use the innovations of AVX-512 in an appropriate way [22]. RLE is the only compression technique tackling uninterrupted sequences of occurrences of the same value, so called runs. In its compressed format, each run is represented by its value and length. Thus, the compressed data is a sequence of such pairs. Our new approach, called RLE512CD, differs significantly from the state-of-the-art vectorized RLE. However, in both cases, there are 4 steps, which are repeated until all elements are processed:

Loading Step: In this first step, the input elements are loaded into a vector register.

Run Detection Step: In the second step, we detect if there are any runs beginning in this register and where they begin or end.

Run Length Detection Step: The run length of the finished runs has to be determined in the third step.

Storage Step: The determined runs are written to memory.

However, there is a plethora of possibilities for the implementation these steps. First, we describe the state-of the-art comparison-based approach. Then, we present our novel CD-based approach.

2.2 State-of-the-Art Vectorization of RLE

Generally, to compress a sequence of integers, the corresponding runs have to be determined and this can be done by comparing each element with its predecessor. If they are equal, a run continues. If they are not equal, a new run starts. These comparisons can be done for more than one element at once using SIMD instructions as shown in [7, 21]. In detail, this state-of-the-art RLE comparison-based vectorization works as follows, whereby the au-

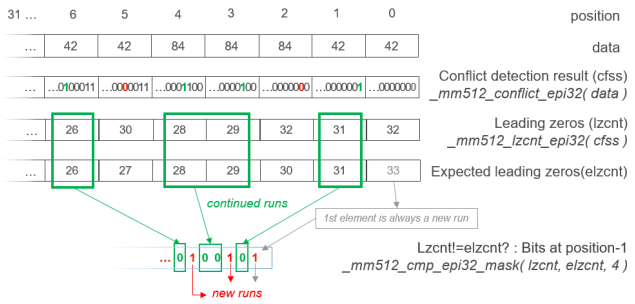

Figure 3: Run detection using *conflict detection* instructions.

Figure 4: Run length determination using *conflict detection* instructions.

thors used 128-bit vector registers (RLE128):

Loading Step 1: One 128-bit vector register v_1 is loaded with four copies of the current input element.

Loading Step 2: The next four input elements are loaded into a vector register v_2.

Run and Run Length Detection Step 1: The intrinsic `_mm_cmpeq_epi32()` is employed for a parallel comparison, so that the four elements in v_1 and v_2 are pair-wise compared at once. The result is stored in a vector register.

Run and Run Length Detection Step 2: Next, a 4-bit comparison mask is obtained using the intrinsic `_mm_movemask_ps()`. Each bit in the mask indicates the (non-)equality of two corresponding vector elements. The number of trailing one-bits in this mask is the number of elements for which the run continues. If this number is 4, then a run's end has not been reached and the execution continues at *Loading Step 2* (new iteration). Otherwise, a run's end is reached that means that run value and run length are appended to the output during the *Storage Step*. The execution continues with *Loading Step 1* at the next element after the run's end (new iteration).

Since only common intrinsics are used, this comparison-based implementation can easily be adapted to 256 and 512 bit-wide registers by loading more elements in wider registers and by using the appropriate intrinsics of AVX2 (256 bit) or AVX-512. Additionally, the *Run and Run Length Detection Steps* can be merged into one step in AVX-512, because there is an intrinsic producing a bitmask directly from the comparison. The corresponding implementations are denoted as RLE256 and RLE512.

2.3 Conflict Detection-based Vectorized RLE

Our novel approach processes every input element only once. Hence, the *Loading Step* always loads 16 new input elements into a 512-bit vector register. The *Run Detection* and *Run Length Detection Steps* are less trivial and are explained in more detail below.

Run Detection In the first sub-step, we create a new vector register containing a **c**onflict **f**ree **s**ub**s**et (*cfss*) of the given source register with the 16 loaded elements using the `_mm512_conflict_epi32` intrinsic. The example in Figure 3 shows the first 7 values of a vector register containing two different values spread over 3 runs. As described above, the newly created vector register consists of 16 bitmasks, where each bitmask shows the equality to all previous elements. However, for detecting a run, it is sufficient to know if the direct predecessor of an element is equal because all elements are either the beginning of a new run or the continuation of another run. If an element is equal to its direct predecessor, the element continues a run. If they are not equal, a new run starts. Hence, only one bit in every bitmask of *cfss* is of interest, i.e. the bit which indicates the equality with the direct predecessor. To find this bit for all elements in parallel, two more operations are necessary:

First (second sub-step), we count the leading zeros of all bitmasks in *cfss* (*lzcnt*). The number of leading zeros should decrease with every element if a run is continued because there is always one more bit set in the subsequent element, e.g. the bitmask at position 1 should have $32 - 1 = 31$ leading zeros, the bitmask at position 2 should have $32 - 2 = 30$ leading zeros and so on. If a run is not continued, the next bit is not set and the number of leading zeros does not decrease. To find out, if the number of leading zeros is decreasing, we compare *lzcnt* with a predefined vector, containing decreasing numbers, for inequality (third sub-step). As shown in Figure 3, this comparison returns 0 for every element which continues a run. Thus, the position of the ones in the final bitmask indicates the position of the start of all runs in this register. Note that the first element always starts a new run.

Run Length Detection Fundamentally, the run length is already encoded in the results of the conflict detection (*cfss*) operation, because each continuous sequence of 1s in the bitmasks indicates a subsequent occurrence of equal numbers. Hence, the number of the most significant subsequent 1s in the bitmask of every last element of a run indicates the length of the run. To get this number, at first the position of the last element of every run has to be determined. This can be done by using the bitmask generated as the result of the run detection (*cfss*). Since every 1 in this bitmask indicates the beginning of a new run, we can get the end of the runs by shifting this mask one bit to the right. Now every 1 indicates the end of a run. Then, the bitmasks at these end positions in the output of the conflict detection (in *cfss*) are selected. In Figure 4, which continues the example from Figure 3, one bitmask is selected as an example. To retrieve the number of subsequent set bits in this bitmask, 3 sub-steps are executed:

1. Shift the elements in the result of `_mm512_conflict_epi32` by the number of leading zeros (leading zeros were derived during run detection). In Figure 4 we shift by 28 bits. Now, the sequence is at the beginning of the bitvector.
2. There is no intrinsic for counting leading 1s, so the result from the previous sub-step is inverted.
3. Then, the leading zeros are counted in the third sub-step. In the example, there are two leading zeros.

Since the bit for the first element of a run is always set to 0 during conflict detection, the result has to be increased by 1. Hence, the run length for the second run is $2 + 1 = 3$. In our implementation, these steps are executed in parallel for all runs by using the intrinsics shown in Figure 4. Before storing the results, it must be checked whether the first run of a register is a continuation of the last run of the previous register. If it is a continuation, the run lengths are added and the run is stored once.

Storage Step Finally, the run values and run lengths must be written back to main memory. For this, there are two possible cases: (a) per integer or (b) per vector. Case (a) represents the output

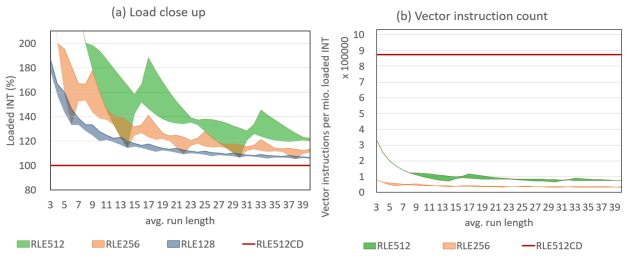

Figure 5: (a) The number of loaded integer as a percentage of the integers in the data set. Only *RLE512CD* shows a constant behavior. (b) The number of vector instructions per million loaded integers (excluding loading and storing) is significantly higher for *RLE512CD* compared to RLE512 and RLE256.

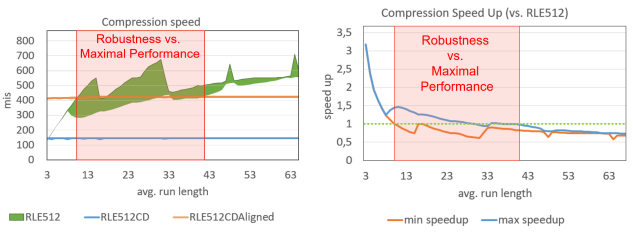

Figure 6: (a) The compression speed for RLE512 varies depending on the run length and the variance of the run length, while our novel implementation shows a constant compression speed. The costs of the scatter store for RLE512CD are clearly visible. (b) The minimum and maximum speedup for RLE512CDAligned compared to RLE512.

format proposed by the state-of-the-art implementation [7], where a sequence of (value, run length)-tuples is stored. An advantage of option (a) is that the output is independent from the vector word size. The disadvantage is that the values and run lengths cannot be loaded sequentially into a vector register again for processing the compressed values, e.g. for aggregating. Case (b) stores sequences of values and run lengths which are as long as a vector word, e.g. 16 values followed by 16 run lengths. Option (b) requires the vector word width as necessary meta data but it is also ideal for processing the compressed data with vector instructions.

2.4 Evaluation

The state-of-the-art implementation includes branches during the loading steps, i.e. depending on the properties of the input data, a different amount of vector loads is executed. Additionally, some input elements are processed more than once, e.g. if a run ends after the first element in *Run and Run Length Detection Step 2*, the remaining 3 elements are loaded again in *Loading Step 1*. To analyze the magnitude of this redundant processing, we counted the load instructions for different average run lengths and all possible variances for each average run length, whereby we used an input sequence with 100 million integers in all experiments. For instance, the maximal variance for an average run length of 5 is ± 4 resulting in the interval $[1, 9]$ for the possible run lengths. Then, we selected the minimal and the maximal number of load instructions and visualized them in Figure 5(a) for RLE128, RLE256, and RLE512. The x-axis shows the average run length and the y-axis shows the number of loaded elements as a percentage of the elements in the input sequence, e.g. 200% means that on average every element is loaded twice. For comparison, we also show the number of necessary vector operations in Figure5(b).

From these experiments, we can conclude that the state-of-the-art RLE uses a significantly higher number of load operations than our novel approach. This effect increases when the bit-width increases and when the run length decreases. Additionally, this overhead is not constant and depends heavily on the data properties. In contrast, the total number of vector operations is lower for the state-of-the-art RLE. Thus, it comes down to the number of loaded and processed integers versus the amount of executed instructions. Depending on the system, this can have different effects on the compression speed. We evaluated our approach on an Intel Xeon Phi KNL 7250. Figure 6 shows the compression speed and the speedup for our approach with vectorwise (RLE512CDAligned) and traditional storing (RLE512CD), and for the state-of-the-art RLE512.

The first obvious finding is that RLE512CD shows an almost constant compression speed as expected. However, the scatter store used in RLE512CD is too slow to compete with RLE512. For RLE512CDAligned, there are 3 different regions: (1) RLE512CDAligned always outperforms RLE512 for very small run lengths (<12). (2) Between the run lengths of 11 and 40, there is no binary decision possible between RLE512 and RLE512CDAligned. RLE512 shows the highest peak performance but also the lowest possible performance. RLE512CDAligned does not reach the peak performance but guarantees a constant compression speed, i.e. it is robust. (3) For run lengths greater than 40, the state-of-the-art implementation always shows the highest compression speed. Hence, at the transitions of these regions, the applied implementation should be changed. Additionally, a decision between maximal peak performance and robustness must be made in region (2). The same regions as for the compression speed can be shown for the speed up in Figure 6(b).

3. DATA LAYOUT CONSIDERATIONS

Generally, the processing of consecutive data elements can be considered as a great challenge to optimize existing or even new algorithms using SIMD. In the context of compression, single data elements are often processed with respect to the surrounding elements, meaning that the distance d should equal one. As mentioned in Section 2, this can be achieved by combining aligned an unaligned loads using a *horizontal data layout*, ending up in an increase of load operations. However, another challenge using the horizontal data layout exists when it comes to inter-register aggregations. Frame-Of-Reference Encoding (FOR) is a prominent example which uses that kind of operation [10, 27]. FOR encodes the difference per element within a fixed size frame with respect to the minimum value of that frame. Thus, the performance of FOR relies on fast aggregation, namely a minimum operation, for a sequence of data. While the latest vector extension AVX512-F supports this kind of operations natively within one vector register, other vector extensions need additional scalar computations. The discussed method utilizing overlapping loads cannot handle that kind of challenge in an efficient way.

To tackle both challenges, a different memory layout seems necessary. Within this layout, consecutive data elements are distributed to the same lane of different vector registers (see Figure 7a) using existing vector operations (see Secction 3.2). We used a nxn matrix where n is the vector size, which holds n vector registers with the corresponding data elements. This approach facilitates a per-element processing within a given frame while preserving data parallelism enabled through SIMD. If the frame size equals the vector size, even stream based processing can be realized.

3.1 Evaluation

The horizontal FOR algorithm reads a fixed amount of data into multiple vector registers, performs the vector operation `min` on all

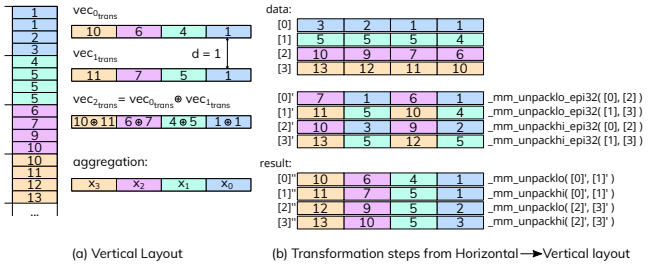

Figure 7: (a) In-vector data organization using a vertical layout (SSE). (b) Transformation from horizontal layout (data[0] - data[3]) to vertical layout (result[0]" - result[3]") using SSE instructions. unpacklo interleaves the low two elements of two given vector registers, unpackhi interleaves the upper two elements respectively.

vector registers resulting in one register containing n explicit minima of the given data. To get the overall minimum the resulting register is transferred to memory and a scalar minimum operation is applied to that memory location. The global minimum of the corresponding frame is broadcasted to a vector register afterwards. The data is loaded into vector registers again and the register containing the global minimum is subtracted from the data. The resulting vectors are stored to memory in a continuous manner.

The vertical FOR algorithm basically performs the same operations until the global minimum has to be retrieved. Instead of transferring the data into a temporary memory location, the global minimum for a corresponding frame is already present within every lane. Thus, the resulting vector register is used to apply the subtraction. Afterwards, the results are stored aligned and continuously to memory.

Comparing the effective throughput of the horizontal and vertical FOR implementation, the performance increases with bigger SIMD registers through the improvement in data parallelism (see Figure 8(a)). The algorithm working with horizontal layouted data needs additional store operations for transferring the processed data from a vector register to an intermediate memory location and a subsequent scalar aggregation. The number of scalar operations grows to the same extent as the vector register size, leading to a lower performance for AVX512 compared to AVX2. An implementation taking advantage of new instructions offered by AVX512 like `__mm512_reduce_min` is considered to be faster than the AVX2 version, but the `reduce` operations where not supported by the used compiler (gcc 7.0.1). The additional overhead from scalar operations is eliminated by an algorithm using vertical layouted data. As a consequence, the vertical algorithm performs better than the horizontal and can even utilize bigger vector registers for improved throughput. From the experiment, we can conclude that a vertical layout can be used to avoid additional scalar operations and increase the performance of certain algorithms. However, most vectorized lightweight data compression algorithms use the horizontal data layout [7].

To evaluate the applicability of the introduced memory layouts, we implemented two FOR algorithm using the horizontal and vertical memory layout respectively. We assume that the input data already has the appropriate layout and the resulting layout correspond with the input layout.

3.2 Integration

Within the field of (column store) in-memory databases, scan and lookup operations are crucial. Previous research has shown that the horizontal layout fits good for lookup operations while a vertical

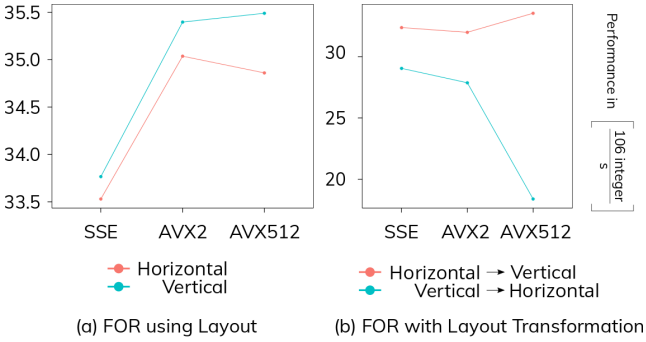

Figure 8: Evaluation of FOR with different variants. (a) FOR using different memory layouts without transformation. (b) FOR transforming the memory layout.

layout should be applied for scans [15]. Typically data is organized either horizontally or vertically within the input and output respectively, leading to an intermediate pre- and postprocessing step for rearranging that data to match the specific layout format. If both layouts should be used side by side, the costs for a transformation between the layouts has to be evaluated. We implemented the already evaluated FOR algorithm in two more variants, performing a straightforward transformation from a horizontal layout (input) to a vertical layout (output) and vice versa.

Variant I (Horizontal → Vertical) reads horizontal layouted data, transforms the data layout within the nxn matrix and stores the result in a vertical format. While the transformation of a given sequence of consecutive data elements (horizontal layout) into the vertical layout is natively supported by AVX2 and AVX512-F (`_mm[256|512]_i32gather`), SSE needs some additional effort. We used 4 aligned loads alongside 4 times `_mm_unpacklo` and `_mm_unpackhi` respectively (see Figure 7b). To store the data an aligned store of the transformed vector registers takes place.

Variant II (Vertical → Horizontal) reads vertical layouted data through an aligned load, processes it and stores the result horizontal layouted. Thus, an element wise random access store from an existing vector register (scatter store) into memory is only supported by AVX512-F, we performed the transformation from the vertical to the horizontal layout completely within vector registers for SSE and AVX2. The SSE implementation uses 4 unaligned stores and 6 times `_mm_unpacklo` and `_mm_unpackhi` respectively. AVX2 operates on two 128-bit vector registers leading to an additional need of 2 `_mm_unpacklo` and `_mm_unpackhi` operations per vector register.

As shown in Figure 8b, Variant I, which uses aligned load and store operation while the transformation overhead take place within vector registers, outperforms Variant II. In general it can be assumed that continuous load and store operations are faster than random access memory operations. This can be seen through the performance penalty for the scatter store (AVX512) of Variant II.

4. FUTURE WORK

The vertical layout meet its limits when it comes to non structure-preserving algorithms like RLE or NS. To apply the vertical layout to this class of algorithms, a selective random access store has to be supported by the underlying vector hardware to transfer discontinuous values into memory. This functionality is only available within the AVX512-F instruction set. Further investigations to evaluate the benefits of a vertical layout for non structure-preserving algorithms seems promising.

5. RELATED WORK

The efficient utilization of SIMD (Single Instruction Multiple Data) instructions in database systems is a very active research field [17, 25]. On the one hand, these instructions are frequently applied in lightweight data compression algorithms [24]. In this domain, null suppression (NS) is the most studied lightweight compression approach, whereby the basic idea is the omission of leading zeros in the bit representation of integers [14]. However, none of these approaches uses the leading zero count intrinsic of the *Conflict Detection* feature set of AVX-512. The application would be very interesting and should be definitely investigated. On the other hand, SIMD instructions are also used in other database operations like scans [9], aggregations [25] or joins [4]. To best of our knowledge, none of these approaches uses AVX-512 CD, although the operations could benefit from CD.

6. CONCLUSION

In this paper, we showed that the further development of existing vector extensions should be investigated in detail to utilize the opportunities to a maximum extent. Thus, we discussed two different approaches, namely the usage of new operations offered by novel instruction sets and different memory layouts depending on the algorithmic task. As an example for the possibilities of new instruction sets we propose an RLE compression algorithm which uses the newly introduced conflict detection instruction set which is available in Intel's AVX-512 vector extension. The performance of our RLE implementation outperforms the state-of-the-art RLE compression algorithm for small run lengths. Furthermore, the performance is constant due the nature of the processed data. However, our new algorithm is not efficient suitable for sequences with long run lengths because too many instructions have to be executed in comparison to the state-of-the-art approach. To deal with concerns arising from wider vector registers like performance penalties of inter-vector aggregations or distances between processed data, we demonstrated a lightweight adaptable vertical memory layout approach to work with consecutive data and even continuous data streams within the given vector registers. Our approach works with state-of-the-art vector extensions, also enabling new possibilities for algorithms from a conceptual point of view. Furthermore we showed, that a transformation from the common horizontal to the vertical layout can be realized completely within given vector registers.

7. REFERENCES

[1] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Softw., Pract. Exper.*, 40(2), 2010.

[3] D. Arroyuelo, S. González, M. Oyarzún, and V. Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *SIGIR*, 2013.

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[5] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[6] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[7] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner. Lightweight data compression algorithms: An experimental survey. In *EDBT*, pages 72–83, 2017.

[8] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12), 2016.

[9] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, pages 31–46, 2015.

[10] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, 1998.

[11] J. Hildebrandt, D. Habich, P. Damme, and W. Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *IMDM@VLDB*, pages 40–56, 2016.

[12] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9), 1952.

[13] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner. ERIS: A numa-aware in-memory storage engine for analytical workloads. In *ADMS*, 2014.

[14] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1), 2015.

[15] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing.

[16] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[17] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[18] M. A. Roth and S. J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3), 1993.

[19] F. Silvestri and R. Venturini. Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, 2010.

[20] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. Simd-based decoding of posting lists. In *CIKM*, 2011.

[21] A. Ungethüm, P. Damme, J. Pietrzyk, A. Krause, D. Habich, and W. Lehner. Balancing performance and energy for lightweight data compression algorithms. In *ADBIS Short Papers*, pages 37–44, 2017.

[22] A. Ungethüm, J. Pietrzyk, P. Damme, D. Habich, and W. Lehner. Conflict detection-based run-length encoding — avx512-cd instruction set in action. In *HardBD-Active Workshop, co-located to ICDE*, 2018.

[23] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6), 1987.

[24] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J. Nie, H. Yan, and J. Wen. A general simd-based approach to accelerating compression algorithms. *ACM Trans. Inf. Syst.*, 33(3), 2015.

[25] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *SIGMOD*, pages 145–156, 2002.

[26] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.*, 23(3), 1977.

[27] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.