Johannes Pietrzyk, Dirk Habich, Patrick Damme, Erich Focht, Wolfgang Lehner

Evaluating the Vector Supercomputer SX-Aurora TSUBASA as a Co-Processor for In-Memory Database Systems

Erstveröffentlichung in / First published in:

Datenbank-Spektrum. 2019. 19(3), S. 183–197. Springer. ISSN 1610-1995.

DOI: https://doi.org/10.1007/s13222-019-00323-w

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

Evaluating the Vector Supercomputer SX-Aurora TSUBASA as a Co-Processor for In-Memory Database Systems

Johannes Pietrzyk[1] · Dirk Habich[1] · Patrick Damme[1] · Erich Focht[2] · Wolfgang Lehner[1]

Received: 14 June 2019 / Accepted: 27 August 2019 / Published online: 9 September 2019

Abstract
In-memory column-store database systems are state of the art for the efficient processing of analytical workloads. In these systems, data compression as well as vectorization play an important role. Currently, the vectorized processing is done using regular SIMD (Single Instruction Multiple Data) extensions of modern processors. For example, Intel's latest SIMD extension supports 512-bit vector registers which allows the parallel processing of 8× 64-bit values. From a database system perspective, this vectorization technique is not only very interesting for compression and decompression to reduce the computational overhead, but also for all database operators like joins, scan, as well as groupings. In contrast to these SIMD extensions, NEC Corporation has recently introduced a novel pure vector engine (supercomputer) as a co-processor called SX-Aurora TSUBASA. This vector engine features a vector length of 16.384 bits with the world's highest bandwidth of up to 1.2 TB/s, which perfectly fits to data-intensive applications like in-memory database systems. Therefore, we describe the unique architecture and properties of this novel vector engine in this paper. Moreover, we present selected in-memory column-store-specific evaluation results to show the benefits of this vector engine compared to regular SIMD extensions. Finally, we conclude the paper with an outlook on our ongoing research activities in this direction.

Keywords Vectorization · NEC SX-Aurora TSUBASA · Column stores · Experimental evaluation · SIMD extension

1 Introduction

In our digital world, efficient query processing is still an open challenge due to the ever-growing amount of data. To satisfy query response times and query throughput demands, the architecture of database systems is constantly evolving [14, 17, 25, 28, 4]. For instance, the database architecture shifted from a disk-oriented to a main-memory-oriented architecture to efficiently exploit the ever-increasing capacities of main memory [1, 16, 19, 33]. This in-memory architecture is now state-of-the-art and characterized by the fact that all relevant data is completely stored and processed in main memory. Additionally, relational tables are organized by column rather than by row [1, 16, 33, 4, 6] and the traditional tuple-at-a-time query processing model was replaced by newer and adapted processing models like column-at-a-time or vector-at-a-time [1, 16, 33, 36, 4].

To further increase the performance of queries, in particular for analytical queries in these in-memory column stores, two key aspects play an important role. On the one hand, data *compression* is used to tackle the continuously increasing gap between computing power of CPUs and memory bandwidth (also known as memory wall [4]) [15, 2, 3, 35, 8]. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy, and less misses in the translation lookaside buffer. On the other hand, in-memory column stores constantly adapt to novel hardware features like *vectorization* using Single-Instruction Multiple Data (SIMD) extensions [32, 36], GPUs [14, 18], or non-volatile main memory [28]. In particular, vectorization is heavily used to

Johannes Pietrzyk
johannes.pietrzyk@tu-dresden.de

✉ Dirk Habich
dirk.habich@tu-dresden.de

Patrick Damme
patrick.damme@tu-dresden.de

Erich Focht
erich.focht@emea.nec.com

Wolfgang Lehner
wolfgang.lehner@tu-dresden.de

[1] TU Dresden – Professur für Datenbanken, Dresden, Germany

[2] NEC HPC Europe GmbH, Stuttgart, Germany

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

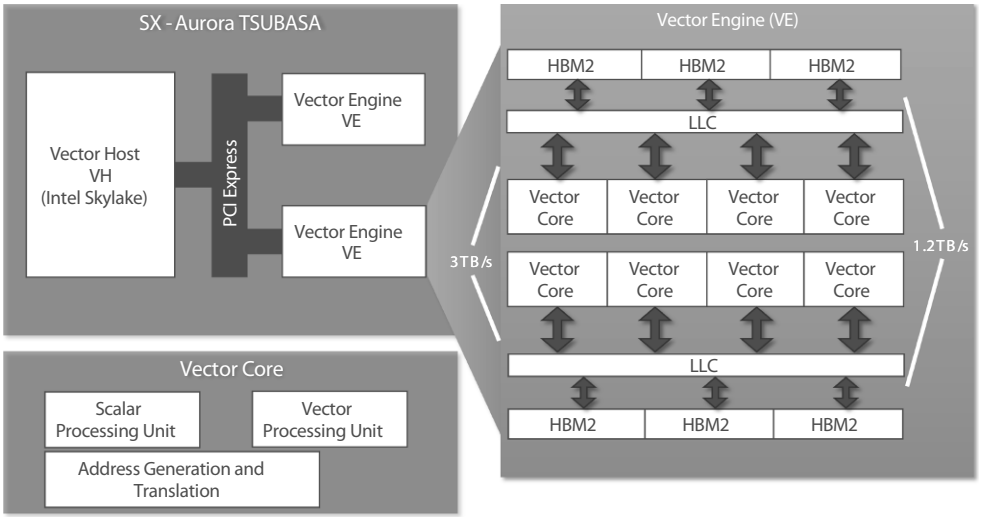

Fig. 1 SX-Aurora TSUBASA architecture

improve the processing performance for regular query operators as well as for compression and decompression by parallelizing computations over vector registers.

Currently, vectorization is performed using regular SIMD extensions of modern processors such as Intel's SSE (Streaming SIMD Extensions), Intel's AVX (Advanced Vector Extensions), or ARM's Neon (Advanced SIMD Extension aka Neon). Up to now, Intel's latest SIMD extension supports 512-bit vector registers corresponding to 8 64-bit values in parallel. In contrast to that, NEC Corporation recently released a pure vector engine called SX-Aurora TSUBASA [20] as a co-processor in a heterogeneous hardware environment. This vector engine features (i) a vector length of §2 KB (16.384 bits) which significantly exceeds the size of regular SIMD extensions—256 64-bit values in parallel—and (ii) the world's highest memory bandwidth of up to 1.2 TB/s per vector processor [20]. From that perspective, this vector engine is very interesting for in-memory database systems. In particular, from the following aspects: (i) In contrast to other co-processors like GPUs, the processing principle does not have to be changed. That means, vector processing in a uniform way can be performed on the CPU as well as on the vector engine. (ii) The vector engine SX-Aurora TSUBASA offers enough memory bandwidth to fill the very large vectors with data for an efficient processing.

Our Contribution and Outline: In this paper, we make the following contributions, whereat this journal paper is an extended version of a workshop paper [29] published at the NoDMC workshop:

1. We start by describing the unique architecture and properties of this novel vector engine SX-Aurora TSUBASA in Sect. 2.
2. Then, we present *MorphStore* [13], a regular in-memory column store system with a novel compression-aware and highly vectorized query processing concept in Sect. 3. We added the system description of *MorphStore* to this journal version to highlight our general research direction in this field.
3. Based on *MorphStore*, we present selected evaluation results to show the benefit of this vector engine compared to regular SIMD extensions of modern processors in Sect. 4. Moreover, we show that our *MorphStore* concepts are well-designed for Intel processor systems as well as for the NEC vector engine.

Afterwards, we briefly introduce our ongoing research activities in Sect. 5. Finally, we conclude the paper with a short summary in Sect. 6.

2 Hardware System SX-Aurora TSUBASA

NEC Corporation has a long tradition in vector supercomputers with a series of NEC SX models starting in 1983. The most recent model is NEC SX-Aurora TSUBASA [20]. In the following sections, we will describe the overall architecture, the vector processing and the programming approach of this novel SX-Aurora TSUBASA model.

2.1 Overall Architecture

The overall architecture of SX-Aurora TSUBASA completely differs from its predecessors. The new system model is a heterogeneous system consisting of a vector host (VH) and one or more vector engines (VE). As illustrated in Fig. 1, the vector host is a regular Intel Xeon Skylake CPU featuring a standard x86 Linux server that provides standard operating system functions. Moreover, the vector host also includes a special operating system for the vector engines called *VEOS* running in the user mode of the vector host.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

Table 1 Specifications for SX-Aurora TSUBASA

Type	Frequency	DP Performance of a core	DP Performance of a Processor	Memory Bandwidth	Memory Capacity	Vector Size in bit
VE 10A	1.6 GHz	307.2 Gflop/s	2457.6 Gflop/s	1228.8 GB/s	48 GB	16.384
VE 10B	1.4 GHz	268.8.2 Gflop/s	2150.4 Gflop/s	1228.8 GB/s	48 GB	16.384
VE 10C	1.4 GHz	268.8.2 Gflop/s	2150.4 Gflop/s	750.0 GB/s	24 GB	16.384
VH Intel Xeon Gold 6126	2.5 GHz	83.2 Gflops/s	998.4 Gflops/s	128 GB/s	96 GB	128, 256, and 512

VEOS controls the vector engines, whereby each vector engine is implemented as a PCI Express card equipped with a newly developed vector processor [20].

As illustrated in Fig. 1 on the right side, each vector processor/engine consists of 8 vector cores, 6 banks of HBM2[1] high-speed memory, and only a last-level cache (LLC) with a size of 16 MB between memory and the vector cores. The LLC is on both sides of the vector cores as depicted in Fig. 1, and it is connected to each vector core through a 2D mesh network-on-chip with a total cache bandwidth of 3 TB/s [20]. Moreover, this vector processor design provides a memory bandwidth of up to 1.2 TB/s per vector engine [20]. Each vector core consists of three core units: (i) a scalar processing unit (SPU), (ii) a vector processing unit (VPU), and (iii) a memory-addressing vector control and processor network unit (AVP). The SPU has almost the same functionality as modern processors such as fetch, decode, branch, add, and exception handling. However, the main task of the SPU is to control the status of the vector core.

2.2 Vector Processing and Specific Systems

This vector engine does not only feature high bandwidths but also a very advantageous architecture of the vector processing units (VPU). Each VPU has three vector-fused multiply add units with 32 vector pipelines and different vector instructions can be independently executed on the units [20]. Generally, the vector length of the VPU is 256 elements[2], each of which is 8 Byte [20]. Thus, one vector instruction executes 256 arithmetic operations within eight cycles [20]. The major advantage, compared to wider SIMD functionalities, e.g., in Intel processors like AVX-512, is that the operations are not only executed spatially parallel, but also temporally parallel which better hides memory latency [20].

Each VPU has 64 vector registers and each vector register is 2 KB in size (256 elements with 8 bytes per element). Thus, the total size of the vector registers is 128 KB per vector core, which is larger than an L1 cache in modern regular processors. To fill these large vector registers with data, the LLC is directly connected to the vector registers and the connection has roughly 400 GB/s bandwidth per vector core [20].

Generally, NEC offers three types of these vector engines called 10A, 10B, and 10C as illustrated in Table 1, which only differ in frequency, memory bandwidth, and memory capacity. In every case, the vector host (VH) is an Intel Xeon Gold 6126 with 12 cores. Tab. 1 also compares VE and VH with respect to double-precision (DP) performance per core or per processor as well as memory bandwidth and memory capacity. As we can see, memory bandwidth of each vector engine is many times higher than that of the vector host, but maximum memory capacity of the vector engine is 48 GB.

The SX-Aurora TSUBASA systems have a high-level configuration flexibility and the series includes three product types:

- A100 is a workstation model with one VH and one VE.
- A300 is a standard rack-mount model with up to eight VEs with one VH. In this case, the maximum size of the vector main memory is 384 GB.
- A500 is designed as large-scale supercomputer with up to eight A300 models connected which results in maximum vector main-memory capacity of 3.072 GB.

With these memory capacities and bandwidths, this heterogeneous system approach is very interesting for memory-intensive applications such as databases.

2.3 Execution Model and Programming Approach

Unlike other accelerators, SX-Aurora TSUBASA is pursuing a different execution model. In general, VE is entirely responsible for executing applications, while VH provides basic OS functionalities such as process scheduling and handling of system calls invoked by the applications on the VE [20]. Applications for VEs are written in standard programming languages such as C, C++ or Fortan without having to use special programming models. In particular, a C library compliant with standards is ported to VE [20]. There-

[1] High Bandwidth Memory Version 2.

[2] In comparison, the vector length of Intel's latest vector extension AVX-512 is limited to 8 elements with 8 Byte per element.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

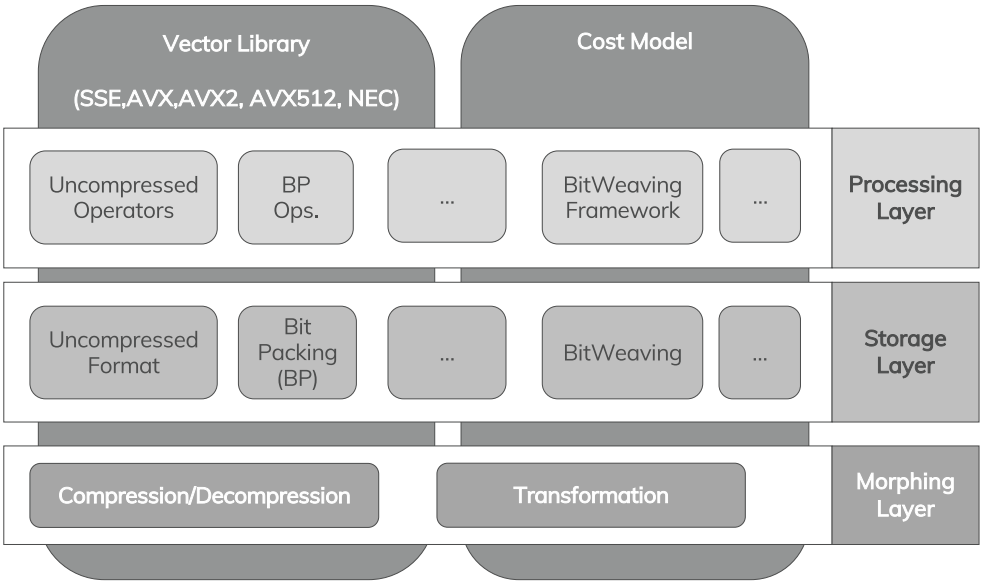

Fig. 2 *MorphStore* architecture [13]

fore, existing (non-vectorized) applications can be ported to VE just by recompiling via the NEC compiler.

3 MorphStore — In-Memory Database System

As already mentioned, *data compression* as well as *vectorization* are extremely relevant techniques for in-memory column-store database systems. As we have shown in [10, 8], there is a large variety of lightweight data compression schemes available and there is no single-best algorithm, but the decision depends on data as well as on hardware properties. However, existing column store systems only provide a very limited set of compression algorithms for base data [1, 11, 22]. Furthermore, during query processing, these systems only keep the data compressed until an operator cannot process the compressed data directly, whereupon the data is decompressed, but not recompressed. Thus, the full optimization potential is not exploited.

To overcome that, we developed *MorphStore*[3], a new regular in-memory column-store with a novel compression-aware and highly vectorized query processing concept [13, 15]. The unique features of *MorphStore* are: (i) compression and vectorization are first-class citizens as depicted in Fig. 2, (ii) support a large variety of lightweight integer compression algorithms and vectorization concepts, (iii) a continuous handling of compression from base data through intermediate results, (iv) a cost-based decision for the best-suited compression algorithm, and (v) morphing intermediates from one to another compression scheme to dynamically adapt the physical representation to the changing data characteristics at query run-time. In the following, we briefly describe the different *layers* as depicted in Fig. 2.

Storage Layer. This layer follows a well-known approach: (i) encode values of each column as a sequence of integers using some kind of dictionary encoding [3] and (ii) apply lightweight lossless integer compression to each sequence of integers resulting in a sequence of compressed column codes [1, 7, 8]. As illustrated in Fig. 2, *MorphStore* does not assume or prefer a specific in-memory storage layout. Instead, it aims to support a large variety of lightweight integer compression *algorithms* and a variety of specific *layouts* for compressed data, e.g., BitWeaving [26]. In principle, these are two different things, but since some compression algorithms also specify a storage layout for the compressed data, the pool of possible layouts becomes even larger. Therefore, this layer focuses on the different layouts for storing uncompressed as well as compressed sequences of integers in-memory. We follow this approach, since, as we have shown in [8], the compression algorithms are always tailored to certain data characteristics and their behavior in terms of performance and compression ratio strongly depends on the data. There is no single-best compression algorithm [8], thus we need a large variety to support all possible data characteristics. For the algorithm selection, we introduced a compression-specific cost model allowing the estimation of the compression ratio as well as the performance in [10].

Morphing Layer. While the *storage layer* focuses on providing different layouts, the *morphing layer* provides an infrastructure for a seamless transition (we call it morphing) from data in a specific layout into another layout. Thus, the different (de)compression algorithms, which are responsible for transforming uncompressed data into the corresponding compressed layout and vice versa, are ma-

[3] https://morphstore.github.io.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

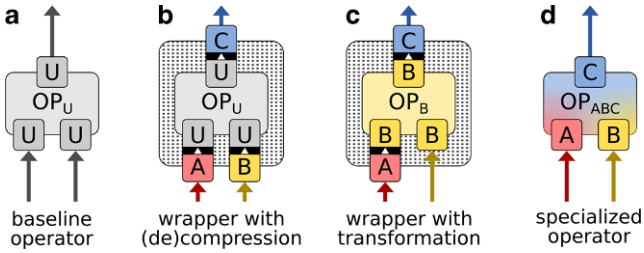

Fig. 3 Integration of compression and operators. *A* to *C* are compressed formats; *U* is uncompressed [7]

jor parts of this layer. Furthermore, we introduced novel transformation algorithms in [9] to directly transform data from a compressed source layout into a compressed target layout. These transformations are also components of this layer.

For our morphing purposes—applying decompression and recompression—*during* query execution, we depend on highly efficient implementations of these existing algorithms. One way to achieve these is to use vectorization allowing the application of one operation to multiple data elements at once. In fact, the employment of SIMD instructions has been the major driver of the research in the lightweight integer compression domain in recent years [12, 24, 34, 8]. To support the different available vector extensions as well as a dedicated vector engine provided by NEC [20, 29] with a low effort, we developed a *Vector Library* abstracting different SIMD extensions that is comparable to [31].

Processing Layer. The execution model of *MorphStore* corresponds to an *operator-at-a-time* model, where all intermediates are materialized in main memory. Thus, this layer provides all physical query operators for *MorphStore*, thereby different degrees of integration between these operators and compression are possible. Figure 3 shows these variants and *MorphStore* supports all of them. The selection of the best-suited operator variant within a query execution plan (QEP) will be done using an appropriate cost model in a subsequent step following the regular query optimization Fig. 3a shows the baseline of processing only uncompressed data.

A first variant to support compressed intermediates is shown in Fig. 3b. The original operator for uncompressed data is surrounded by a wrapper, which temporarily decompresses the inputs and recompresses the outputs. This approach is called *transient decompression* and was proposed in [5], but to the best of our knowledge, it has never been investigated in practice. For efficiency, in *MorphStore* the decompression (recompression) does not work on the entire inputs (outputs), but on small chunks fitting into the L1 cache. Changing the compressed format of the intermediates is possible by configuring the wrapper's input and output formats accordingly. The advantage of this variant is its simplicity: It reuses the existing operator and relies only on n already existing (de)compression algorithms. However, it does not exploit the benefits of working directly on compressed data.

The second variant is to adapt the operator such that it can work *directly* on compressed data (Fig. 3c). Existing works such as [23, 26] have already proposed *certain* operators on *certain* compressed formats. We contribute to this line of research by covering the formats of recent vectorized compression algorithms. For this variant, we assume a common compression format (format B in Fig. 3c) for all inputs and outputs of the operator; for arbitrary combinations of formats, the operator is again wrapped. However, in this case the wrapper utilizes the *direct transformation* algorithms we developed. The idea of bringing compressed inputs into a common format has already been proposed in [23], but only for joins on dictionary encoded data – and without *direct* transformations. This approach requires n variants of the operator and n^2-n transformations, whereby the latter can be reused for all other operators. Nevertheless, the existence of a wrapper still causes a certain overhead. The final variant maximizes the efficiency by tailoring the operator to a specific combination of formats (Fig. 3d). Unfortunately, this approach implies the highest implementation effort, requiring n^{i+o} operator variants.

4 Comparative Evaluation

Based on our *MorphStore* design concepts, we present selective comparative evaluation results to show the benefits of the vector engine compared to regular SIMD extensions of modern processors in this section. Thus, we start with an introduction of our investigated operations in Sect. 4.1, followed by a description of our experimental setup and methodology in Sect. 4.2. Then, we present single-threaded and multi-threaded evaluation results in Sects. 4.3 and 4.4. Finally, we summarize lessons learned from this comparative evaluation in Sect. 4.5.

4.1 Selected Investigated Operations

Since all *MorphStore* operations are memory bound, we basically have focused our comparative evaluation on examining the specified memory throughput. Therefore, we measured plain sequential memory access in a first step, followed by (i) a compression algorithm and (ii) a column-scan which can be considered as a fundamental physical query operator.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

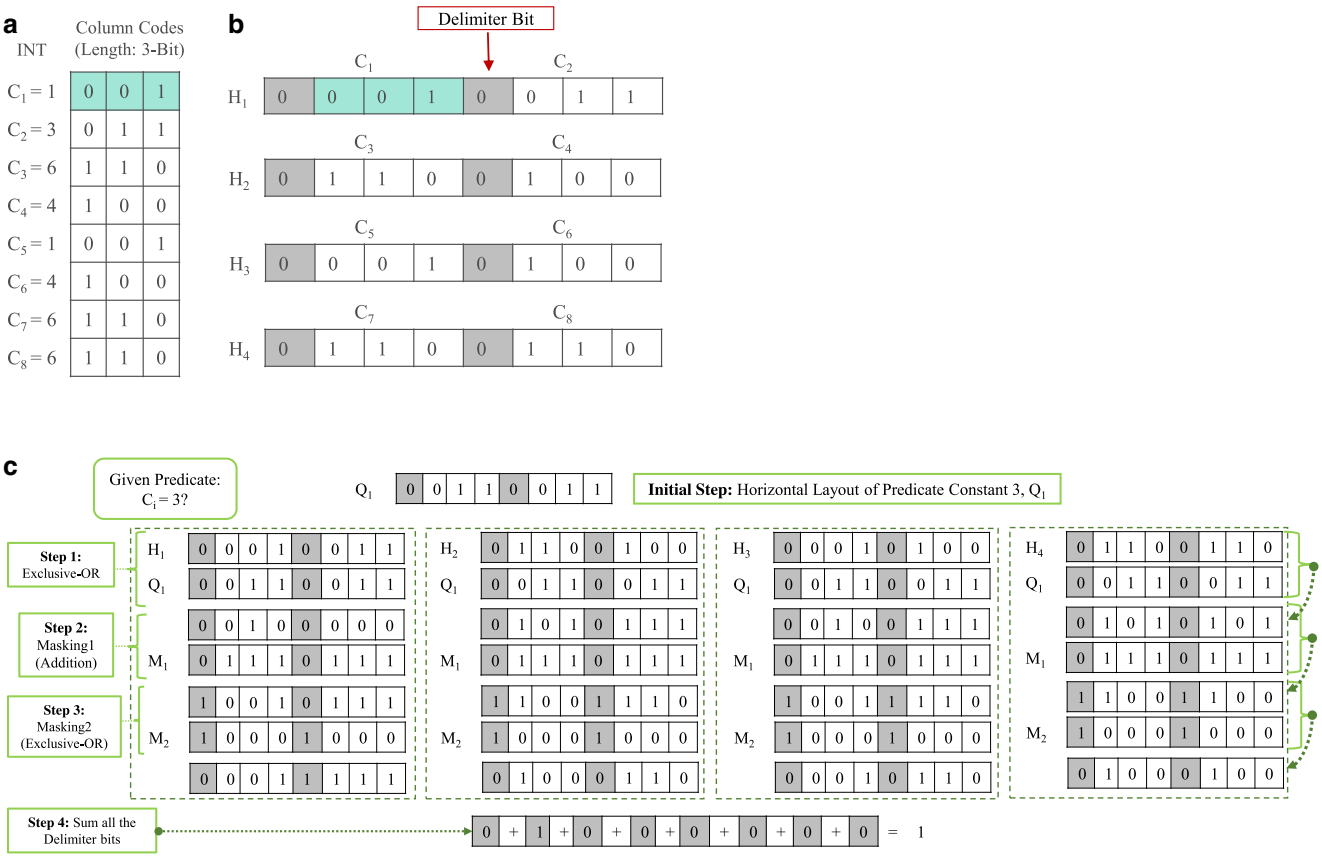

Fig. 4 Illustration of BitWeaving/H [27]. **a** Data, **b** BitWeaving/H, **c** Predicate Evaluation

4.1.1 Memory Access Primitives

Every physical *MorphStore* query operator has the same operation mode: (i) reading data from memory, (ii) processing data, and (iii) writing the result back to main memory. Thus, memory accesses are fundamental tasks in our overall concept and we focus on these core primitives (reading, writing, and copying) as a first step in our evaluation. While reading from memory without any further operations will most probably be deleted by the compiler, an aggregation is performed over the read memory using the bitwise *OR* operator. Thus, only a cache-resident value has to be updated per actual read. Given a relatively fast aggregation operation with regard to the memory access, it can be assumed that the measured throughput is not distorted by computation efforts. To measure the behavior of write-intense sequential memory access, we filled an array with a constant value, like a *memset*. As a combination of reading and writing, we measured the throughput of copying the values from a given array into another array.

4.1.2 Bit Packing Compression

Lightweight integer compression plays an important role in *MorphStore*. Thus, we decided to evaluate a representative compression algorithm called *Bit Packing (BP)* [24]. The basic idea of BP is to partition a sequence of integer values into blocks and compress the values within each block separately. The number of bits used to represent every value in a block is determined by the effective bit width of the largest value in that block. The compression operations consist of the following steps.

Step 1: Partition sequence of integer values into blocks.
Step 2: Read values in each block to determine the bit width of the largest value in the block.
Step 3: Read the values again for bit packing based on the largest bit width found in the previous step.
Step 4: Write packed words to output.

The block length depends on the used vector length [10, 24]. For example, for a vector length 128-bit, the number of integers per blocks has to be 128 to get an aligned output [24]. This means for the VE, each block contains 16.384 integer values.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

a C++ Code for Aggregation

```
1  /*...*/
2  #pragma omp parallel
3  {
4  #pragma _NEC vreg(aRes)
5  #pragma omp for
6  #pragma _NEC noouterloop_unroll
7    for(;i<nBuf;i+=256){
8  #pragma _NEC shortloop
9      for(j=0;j<256;++j){
10       aRes[j]|=aSrc[i+j];
11     }
12   }
13   for(k=0;k<256;++k){
14     nRes|=aRes[k];
15   }
16 }
17 /*...*/
```

b nc++ Listing after compilation

```
1  LINE DIAGNOSTIC MESSAGE
2     1: Vector reg. assigned.: aRes
3     2: Parallel routine generated.
4     7: Parallelized by "for".
5     9: Vectorized loop.
6    13: Vectorized loop.
7    14: Idiom detected.: Bit-op
8  /*...*/
9  LINE LOOP STATEMENT
10 /*...*/
11    7- P---> for(;i<nBuf;i+=256){
12    8- |       #pragma _NEC shortloop
13    9- |V---->   for(j=0;j<256;++j){
14   10- ||  V      aRes[j]|=aSrc[i+j];
15   11- |V---     }
16   12- P---    }
17   13- V---> for(k=0;k<256;++k){
18   14- |   V   nRes|=aRes[k];
19   15- V---> }
20 /*...*/
```

Fig. 5 Excerpt of C++-code for read-intense task and corresponding diagnostic listing, produced by nc++ while compilation with optimization indications. **a** C++ Code for Aggregation, **b** nc++ Listing after compilation

4.1.3 Column Scan Operator

As a representative physical query operator, we evaluated a column scan operation using a state-of-the-art approach called BitWeaving/H [26] (operator according to Fig. 3d). Fundamentally, BitWeaving assumes a fixed-length order preserving compression scheme (like bit packing), so that all compressed column codes of a column have the same bit length [26]. Then, the bits of the column codes are aligned in main memory in a way that enables the exploitation of intra-cycle parallelism using ordinary processor words. An example is shown in Fig. 4a, where eight 32-bit integer values C_i are represented using 3-bit compressed column codes. As illustrated in Fig. 4b, the column codes are contiguously stored in processor word H_i in BitWeaving/H, where the most significant bit of every code is used as a delimiter bit between adjacent column codes. In our example, we use 8-bit processor words, so that two 3-bit column codes fit into one processor word including one delimiter bit per code. The delimiter bits are used later to store the predicate evaluation result.

Now, the task of a column scan is to compare each column code with a constant C and to output a bit vector indicating whether or not the corresponding code satisfies the comparison condition. To efficiently perform such a column scan using the BitWeaving/H, Li et al. [26] proposed an arithmetic framework to directly execute predicate evaluations on the compressed column codes. There are two main advantages: (i) predicate evaluation is done without decompression and (ii) multiple column codes are simultaneously processed within a single processor word using full-word instructions (intra-cycle parallelism) [26]. The supported predicate evaluations include equality, inequality, and range checks. For each of these predicate evaluations, a function consisting of arithmetical and logical operations is defined [26].

Figure 4c highlights the *equality* check in an exemplary way; the other predicate evaluations work in a similar way. The input from Fig. 4b is tested against the condition $C_i = 3$. Then, the predicate evaluation steps are as follows:

Initially: All given column codes and the query constant number *3* are converted into the BitWeaving/H storage layout (H_1, H_2, H_3, H_4) and Q_1, respectively.

Step 1: An *Exclusive-OR* operation between each word (H_1, H_2, H_3, H_4) and Q_1 is performed.

Step 2: *Masking1* operation (*Addition*) between the intermediate results of Step 1 and the M_1 mask register (where each bit of M_1 is set to one, except the delimiter bits) is performed.

Step 3: *Masking2* operation (Exclusive-OR) between the intermediate results of Step 2 and the M_2 mask register (where only delimiter bits of M_2 are set to one and the rest of all bits is set to zero) is performed.

Step 4 (optional): Add delimiter bits to achieve the total count (final result).

The output is a result bit vector, with one bit per input code that indicates if the code matches the predicate on the column. In our example in Fig. 4, only the second column code (C_2) satisfies the predicate which is visible in the resulting bit vector.

4.2 Experimental Setup and Methodology

All our selected operations were measured on two different versions of the SX-Aurora TSUBASA co-processor (VEs)

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

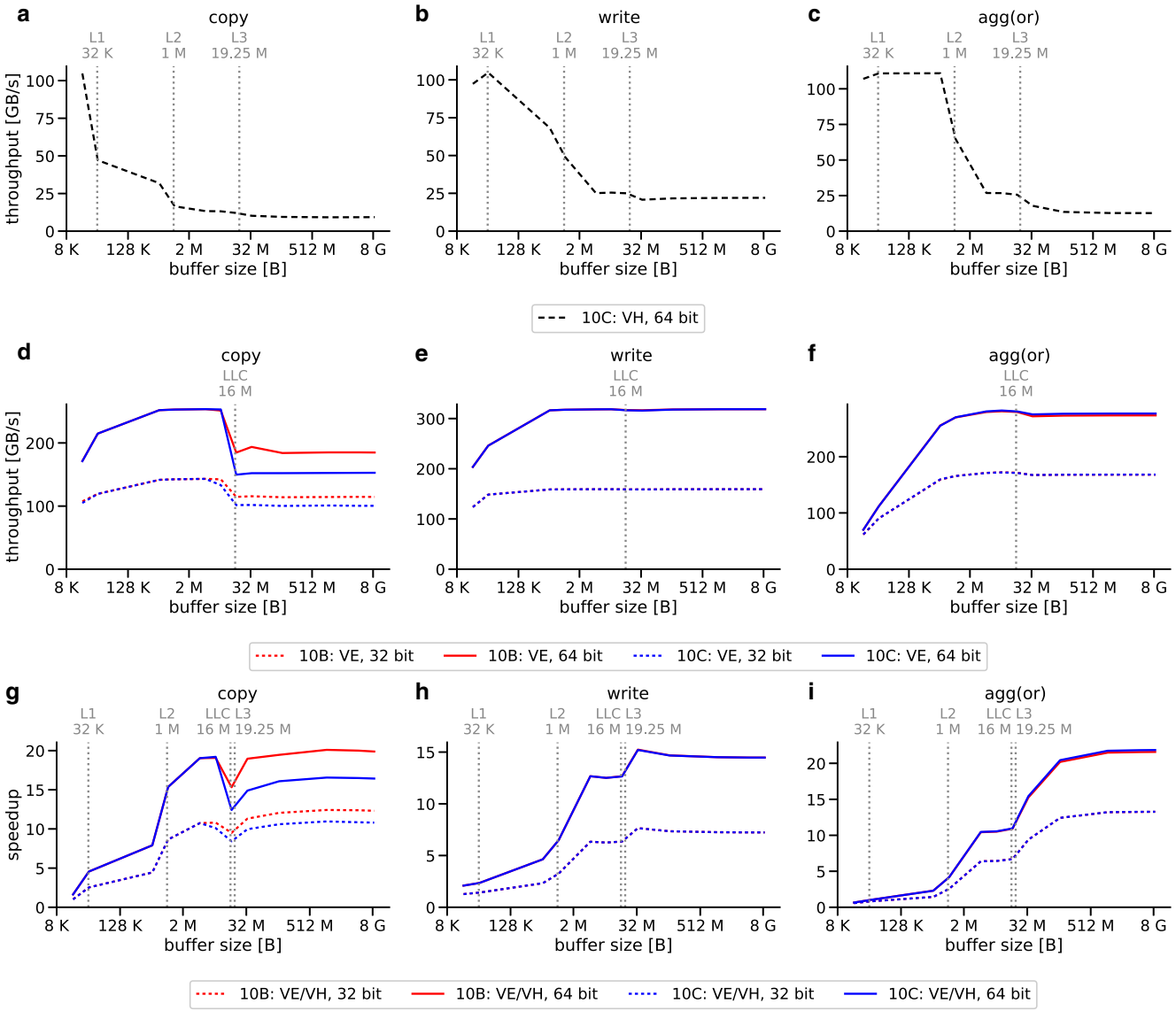

Fig. 6 Single-thread sequential memory access evaluation: measured throughput of VH and VE as well as the speedup obtained by the VE of different IO-operations executed using one thread

and on a recent Intel Skylake processor (VH). The general specifications of these hardware systems are denoted in Table 1. To compile the implemented operators for VH, a gcc 7.3.1 was used with the optimization flags *-O3 -flto* and disabled auto-vectorization with *-fno-tree-vectorize*. For VE, the proprietary NEC compiler nc++ 1.6.0 was used with the optimization flag *-O3 -fipa* and *-mvector*, enabling the auto-vectorization. A distinction between single-thread performance and multi-thread performance was made by linking the binary with and without OpenMP. To minimize the runtime overhead through dynamic linking, all files were linked statically.

The time measurements were performed using a C++ wall-time clock on the VH and inline assembly for retrieving user clock cycles on the VE. As an experiment consists of a measurement of all specified tasks, every experiment was repeated 10 times and the reported runtimes were averaged. To avoid distortion by the actual time measurement, all tasks were repeated multiple times and the accumulated time was divided by the number of repetitions.

Thus, our focus was on evaluating the computing performance of vectorized code alongside the memory bandwidth, all tasks were implemented using vectorization. This was done using intrinsics for the VH. To examine the best performance of the different SIMD extensions offered by the VH, all tasks were implemented and tested using either SSE (128), AVX2 (256) or AVX512.

As shown in Fig. 5a, a combination of compiler specific preprocessor pragma directives, strip mining, and local buffers were used for the VE to facilitate the auto-vector-

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

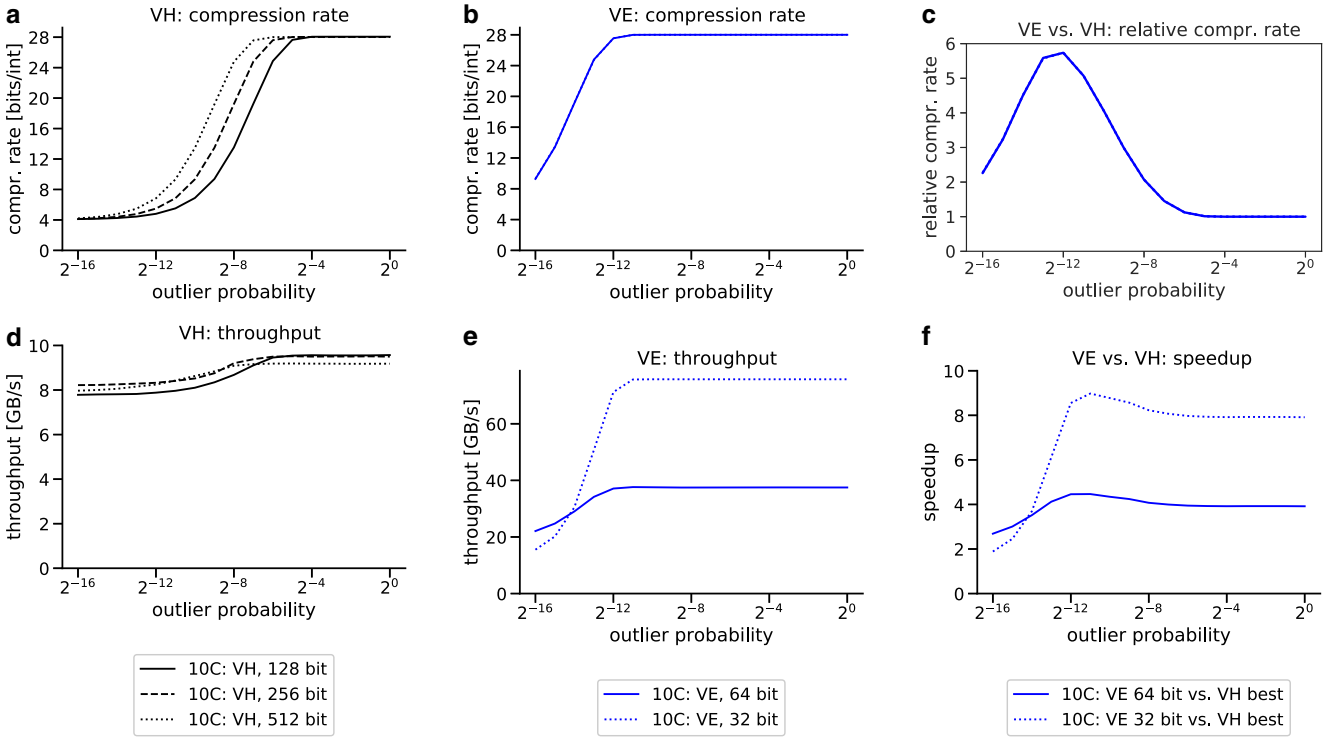

Fig. 7 Single-thread BP compression evaluation: measured compression rate of BP on VH and VE for different outlier probabilities

ization feature of the NEC compiler. At first, the main loop which iterates over the buffer as a whole is fragmented into strips according to the size of a vector register (see Line 7). To prevent the compiler from undoing the so called loop strip-mining, an according pragma was used (see Line 6). The most inner loop, containing the operator-specific instructions, is marked as a shortloop (see Line 8) giving the compiler a hint that the loop should be completely transformed into vector code. In addition, the specific instructions work on temporal buffers which are forcedly assigned to a vector register using *#pragma _NEC vreg(arrayName)* (see Line 4). First measurements showed that this specific hint can significantly improve the performance of the algorithm. OpenMP were introduced through parallel regions using *#pragma omp parallel* (see Line 2) alongside with loop parallelism using *#pragma omp for* depicted at Line 5. When compiling the annotated C++-Code using the NEC compiler, a diagnostic listing, indicating all applied optimizations, can be generated (see Fig. 5b).

Within the VE, two different element sizes (32-, 64-bit) were measured. Within the VH, different load and store modes (stream, aligned, unaligned) were measured. To evaluate a possible influence of different memory structures on the performance, every experiment was executed while processing different sizes of data ranging from 16 KB to 8 GB.

4.3 Single-Thread Evaluation Results

In this section, we present our single-threaded comparative evaluation results for our selected operations.

Memory Access Primitives: Fig. 6 shows the throughput of plain memory access measured on the VH (a)-(c) as well as on the VE (d)-(f). Using the VH, a maximum throughput of around 100 GB/s was obtained when the processed data fits completely into L1. In general, the performance is decreasing when the buffer sizes exceed the cache sizes. While read-intense tasks can utilize L2 without significant performance penalties, write-intense tasks suffer from accessing higher levels of cache. Conversely, the throughput measured on the VE improves with bigger buffer size obtaining an overall maximum throughput of around 300 GB/s (processing 1 MB) for write-intense tasks (see Fig. 6e) and 250 GB/s (processing 2 MB) while executing read-intense tasks (see Fig. 6f). Accessing the HBM2 leads to a marginal decrease in measured throughput.

As shown in Fig. 6d, only the performance of a vectorized copy drops dramatically when the processed data sizes exceed the boundaries of the existing last-level cache (LLC). Since the tasks were executed in vectorized form and a single vector register can hold up to 2 KB data, small buffers prevent the VE from using the given vector registers in an efficient manner. In general, the experiments have shown that processing 64-bit-wide elements lead to significantly higher throughputs compared to the results when

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

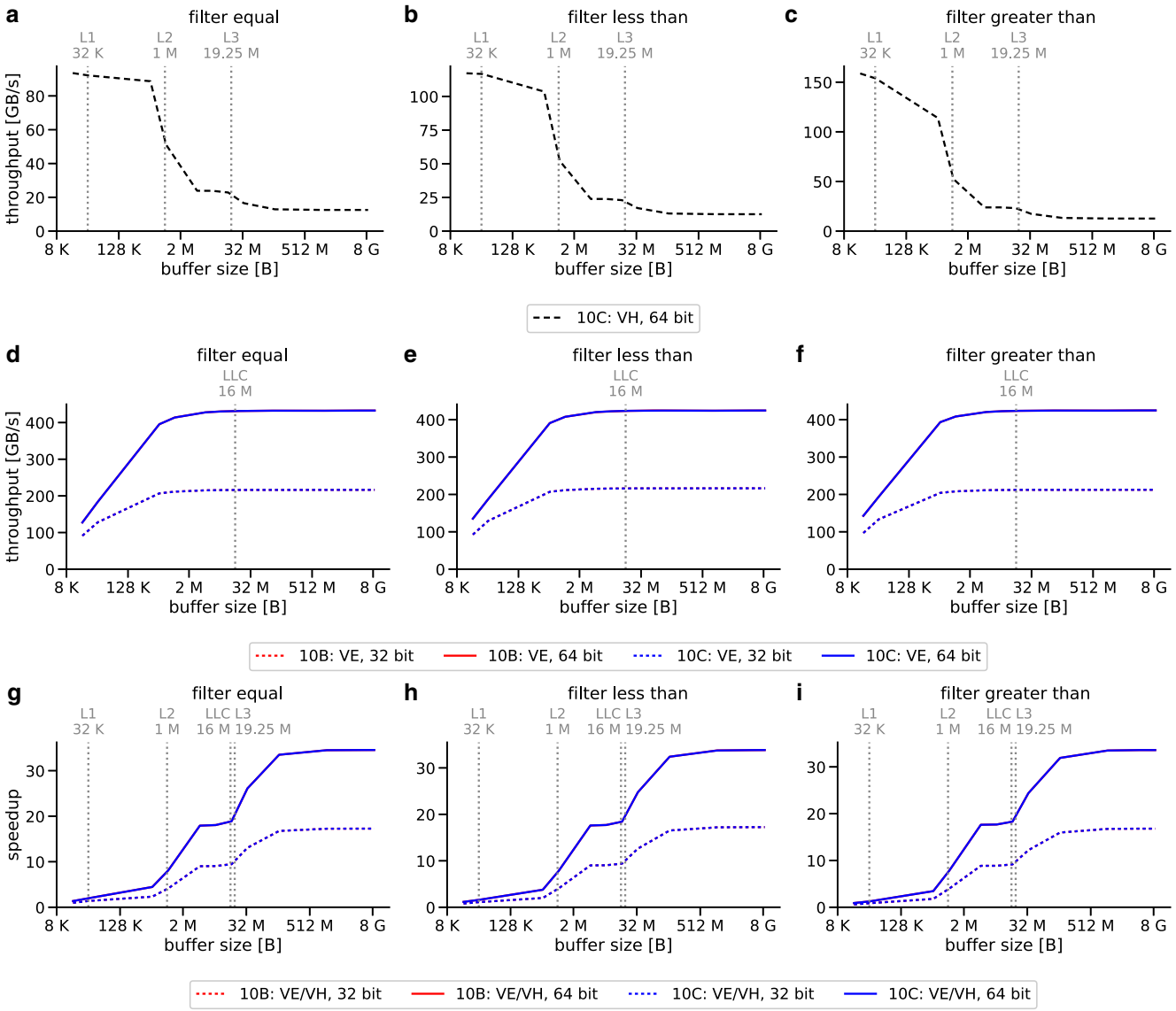

Fig. 8 Single-thread column scan evaluation: measured throughput of VH and VE as well as the speedup obtained by the VE of different BitWeaving/H-operations executed using one thread

working on 32-bit-sized data. This results from the underutilization of available vector registers. The vector pipeline processes its elements at a granularity of 64-bit.

If only 32-bit-wide elements were processed, the remaining bits left unused. An improvement in terms of the memory access could not be achieved on the formally faster TSUBASA 10B neither for vectorized write-intense nor for read-intense tasks. Only the performance of the copy task could benefit from the better memory bandwidth of the 10B. As shown in Fig. 6g–i, both VE outperform the VH for write-intense tasks up to a factor of 15 on the 10C and 20 on the 10B, respectively. A maximum speedup of around 21 was obtained for read-intense tasks when the processed data exceeds the cache and has to be loaded from DRAM (VH) or HBM2 (VE), respectively.

Bit Packing Compression: As mentioned in Sect. 3, one possible opportunity of enhancing the processing speed of query execution is the compression of the underlying data. While performance can be considered as the main focus for primitive operators, compression algorithms also takethe compression rate into account. BP compresses a block with a fixed size, determined by the used vector length. The biggest bit width of any data within this block is used to encode every element. Consequently, the size of the resulting compressed block depends on the biggest value. Taking this into account, we generated different data sets with varying data sizes as well as with varying probabilities of big values (outliers). The results shown for data set sizes of 8 GB are depicted in Fig. 7. As shown in Fig. 7a–c, the number of used bits per integer value (compression rate) increases

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

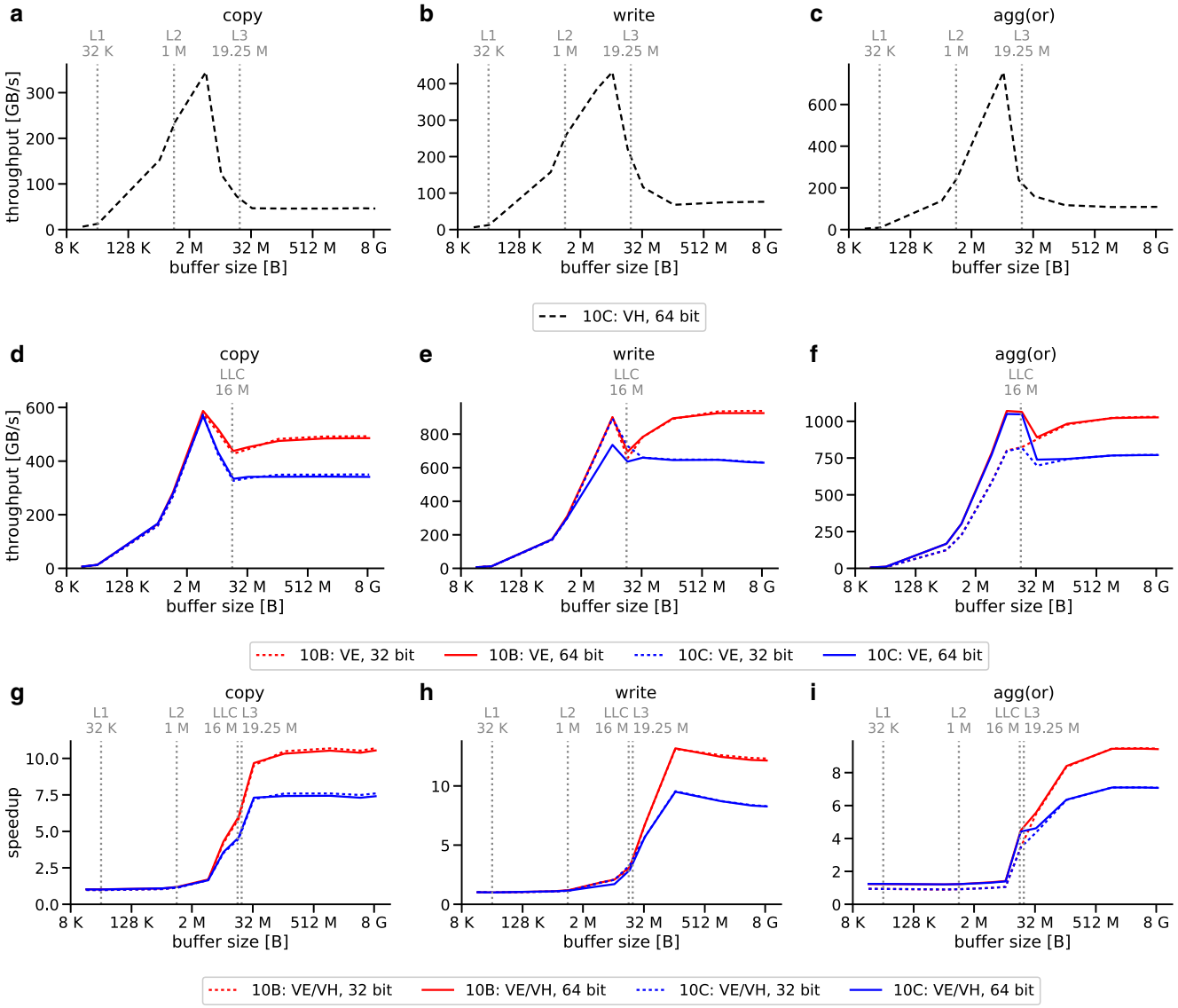

Fig. 9 Multi-thread sequential memory access evaluation: measured throughput of different IO-operations executed on the VH using multiple threads

when the outlier probability gets higher. When bigger vector registers are used, the overall block size increases. This leads to a higher outlier proneness (see Fig. 7a–b). While the vector host support a maximum vector length of 512-bit, the VE processes data blocks of 16.384 elements. Consequently, the compression rate of the vector engine gets significantly higher for small outlier probabilities compared with the VH. When it comes to processing speed, the VE outperforms the VH by a factor of 8. While the VH reaches a maximum throughput of around 9 GB/s, the VE achieves up to 70 GB/s when processing 32-bit values, 30 GB/s when processing 64-bit values, respectively. The reasonably low throughput on the VE results from the fact, that the existing vector pipeline can not be kept busy. A single block contains 16.384× 64-bit values. Thus, only 128 KB data is processed per iteration. After every iteration, a scalar part which is choosing the next bit width has to be executed.

Column Scan Operator: As mentioned in Sect. 4.1, a recent column scan is executed using arithmetic operations. While a filter for *equality* needs two bitwise operators (*XOR* and *NOT*) and an addition, a filter for *less than* requires only one bitwise operator (*XOR*) and an addition. To scan for elements which are greater than the predicate only one addition is executed. These characteristics can be seen in Fig. 8a–c, where this column scan operator achieves a maximum throughput in the range of 80 GB/s (Fig. 8a) up to 150 GB/s (Fig. 8c) when the processed data fits entirely into L1. Running onto the VE, the total amount of executed operations does not affect the overall throughput leading in most cases to a similar behavior as the write task.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

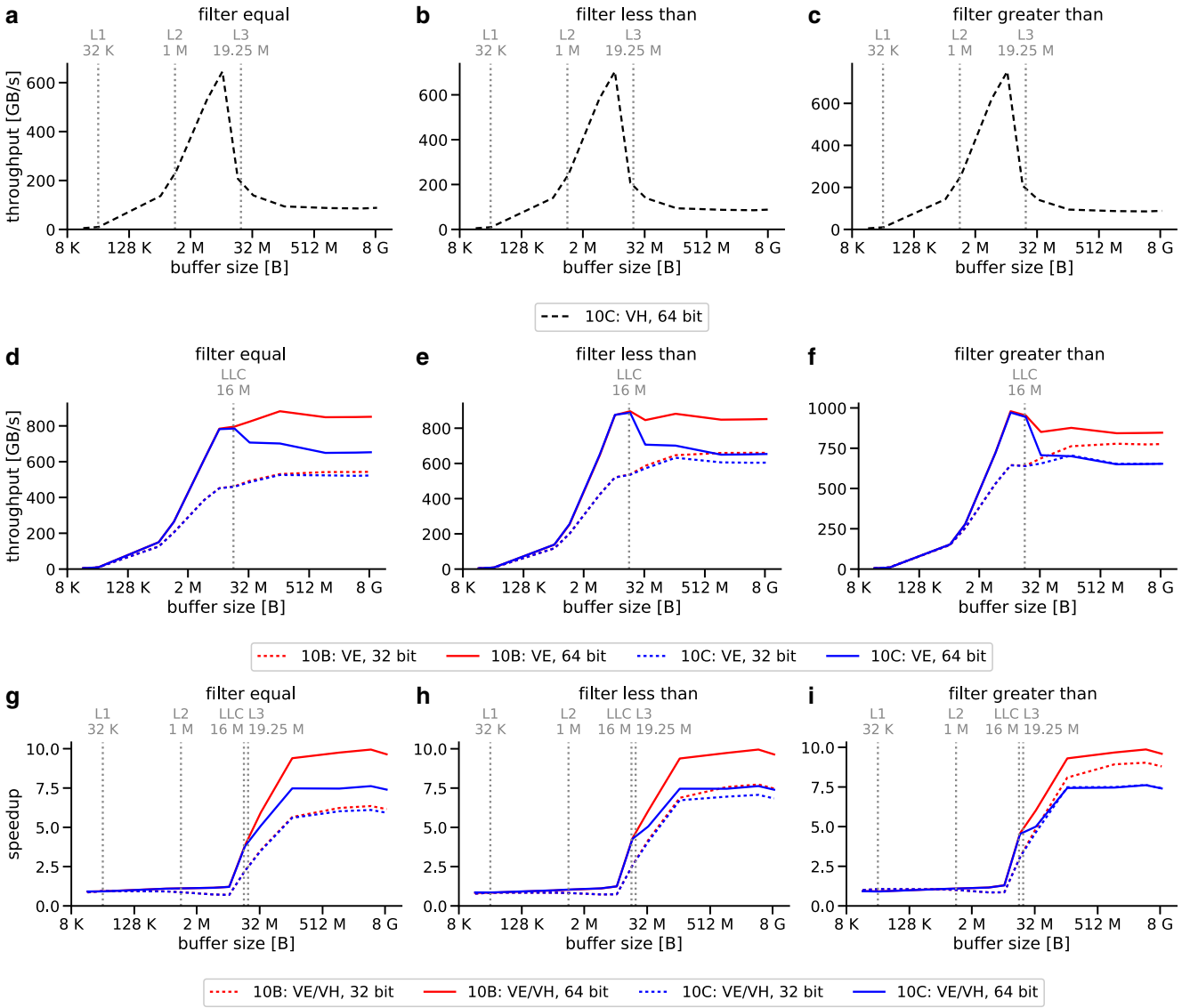

Fig. 10 Multi-thread column scan evaluation: measured throughput of different Bitweaving-H operations executed on the VH and the VE using multiple threads

A maximum throughput of around 400 GB/s was reached when the processed buffer exceeds the boundaries of the LLC. As shown in Fig. 8g–i, both VE outperform the VH up to a factor of 20 when the processed buffer is resident in the caches. When the buffer exceeds the LLC, a speedup of factor 33 could be achieved by the VE.

4.4 Multi-Thread Evaluation Results

Generally, the utilization of multiple threads introduces additional complexity in terms of thread creation as well as data partitioning.

Memory Access Primitives: As shown in Fig. 9, this overhead leads to significant lower throughputs for both the VH and the VE when processing small buffers. Nevertheless, multithreading pays off on the VH when the processed data exceeds the L2 cache (see Fig. 9a–c) obtaining a maximum throughput of around 320 GB/s for copying, 400 GB/s for writing, and 700 GB/s for aggregating, respectively. Taking this into account, using multiple threads for plain memory access can speedup the processing up to a factor of 7 compared to single-thread execution.

The same observation holds for the VE in terms of processing small buffers up to 2 MB. While plain memory access using a single thread reaches a maximum throughput when processing 1 MB and more, multiple threads reach a local maximum of around 4 MB buffer size for copying and 8 MB buffer size for reading and writing, respectively. For bigger buffers which still fit into the LLC, the throughput decreases are probably causes by cache pollution, but

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

we are not able to measure it. As shown in Fig. 9d–f, the measured throughput of the 10C remains stable with a throughput around 300 GB/s for copying and 600 GB/s to 700 GB/s for write- and read-intense tasks, respectively, when the processed buffer size exceeds the boundaries of the LLC. The 10B even outperforms the 10C when processing HMB resident buffers through the higher maximum bandwidth resulting in an overall maximum throughput of around 800 GB/s for writing and nearly 1 TB/s for aggregation (read-intense task). Using multiple threads, the VE 10B outperforms the VH up to a factor of 10 for copying and aggregating, 13 for writing, respectively.

Column Scan Operator: Executing bit-parallel column scan using multiple threads shows similar behavior as the reading task by exceeding the reached throughput of single thread execution by a factor of around 2. Interestingly, the bit width has a significant influence when running on the VE. As shown in Fig. 10d–f, processing 64-bit-wide elements led to higher throughputs in general. This impact of processed word size decreases for processing big buffers using less operations. Using multiple threads, the VE 10B outperforms the VH up to a factor of 10 for the column scan operator as depicted in Fig. 10g–i.

4.5 Summary

Our conducted comparative evaluation has shown that the vector co-processor SX-Aurora TSUBASA can, on the one hand, improve the performance of computational-bound algorithms through the utilization of wide vector registers alongside an efficient vector processing pipeline. On the other hand, memory-bound algorithms can benefit from the integrated high-bandwidth memory in combination with the shared LLC which is accessible from the vector processing units (VPU) directly. Thus, our *MorphStore* concepts are well-designed for Intel systems with their SIMD extensions and for the NEC vector engine SX-Aurora TSUBASA. Generally, we are able to fully utilize the maximum achievable bandwidth on the VH as well as on the VE in a multi-threaded environment with our approaches, however, the VE outperforms the VH.

5 Future Work

Our research results have shown that the NEC vector engine is very beneficial for an efficient in-memory data processing mainly due to the large vector registers as well as the pipeline-based processing model, which perfectly suits data-intensive operations and in-particular to our *MorphStore* design concepts. Compared to standard SIMD extensions of common processors, the NEC vector processor outperforms these extensions by an order of magnitude in average. Thus, we are enthusiastic to continue and to expand our research in this direction.

In particular, we did not present the execution times of complete queries in our comparative evaluation, because we are currently not able to keep all vector registers (vector pipelines) busy during the query evaluation over the different physical query operators of a query leading to a poor performance. For example, a filter operator disqualifies vector elements depending on the predicate and these disqualified elements cause underutilization in the subsequent operators leading to a sub-optimal performance (underutilization of vector registers during query evaluation). Unfortunately, such filter operators are usually executed early to reduce the amount data being processed by later operators. To tackle that challenge, several approaches are possible. One approach would be, for example, adaptively populating vectors during query processing with new data elements as proposed in [21]. This requires the recognition of underutilized vectors within query operators and an efficient approach to reload new data elements for refilling. An alternative approach would be to move such operators to the end of a query data flow in order to reduce the impact of such operators. In our ongoing research activities, we will examine the pros and cons of such (non-standard query) optimizations to optimally reflect the characteristics of the vector engine.

Another interesting research direction would be the efficient utilization of the heterogeneity of the SX-Aurora TSUBASA. As we have already shown in [17], heterogeneous hardware systems provide a great opportunity for database systems to increase the overall query performance if the different processors can be utilized efficiently. To achieve this goal, the main challenge is to place the right work on the right processing unit. Thus, we want to investigate the following research hypothesis: As of now, a CPU-based system *drives* the execution of database queries and offloads specific operations to accelerators like GPUs or even FPGAs. In this context, we are eager to explore the research hypothesis to let the NEC vector processor completely *own* the execution and offload parts of the query evaluation to the CPU host, for example, to conduct complicated logic using SIMD in combination with the many threads of the CPU. Such a *functional accelerator model* has never been explored before in the context of database query processing.

In detail, we want to develop a novel concept to offload non-existing, but desirable vector operations to the SIMD functionality of the vector host. For example, Intel's latest SIMD extension includes a new instruction feature set called *Conflict Detection (CD)*, which allows the vectorization of loops with possible address conflicts. In [30] and [34], we have shown that this CD functionality can be efficiently used to speedup compression as well as hashing.

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

In particular, hashing is a core primitive for the grouping and join operator. With this functionality offloading, we will establish a generic, but database-centric approach to export very specific functionalities to the vector host. Here, the biggest challenge is to design and to implement an efficient approach that seamlessly integrates with pipeline-based processing on the vector engine. Finally, this will yield a deep understanding and experimental evaluation of the interplay between vector engine and vector host.

6 Conclusion

In this paper, we introduced the recently released pure co-processor vector engine NEC SX-Aurora TSUBASA by describing the architecture and the unique properties. This vector engine features a vector length of 16,384-bit with the world's highest bandwidth of up to 1.2 TB/s. Moreover, we presented *MorphStore*, a new regular in-memory column store database system where *compression* and *vectorization* are first class citizens. Based on that design concepts, we described selective comparative evaluation results showing the benefits of this novel vector engine compared to regular SIMD extensions of modern hardware. As we have presented, this vector engine outperforms regular SIMD extensions. Nevertheless, our *MorphStore* design concepts are well-suited for Intel systems as well as the vector engine in a uniform way. Finally, we described our future work and closed paper with a short summary.

Funding This work was funded by NEC Corporation within the project *Highly vectorized query processing on compressed columnar data*.

References

1. Abadi D, Boncz PA, Harizopoulos S, Idreos S, Madden S (2013) The design and implementation of modern column-oriented database systems. Found Trends Databases 5(3):197–280
2. Abadi DJ, Madden S, Ferreira M (2006) Integrating compression and execution in column-oriented database systems. In: SIGMOD, pp 671–682. ACM: New York
3. Binnig C, Hildenbrand S, Färber F (2009) Dictionary-based order-preserving string compression for main memory column stores. In: SIGMOD, pp 283–296. ACM: New York
4. Boncz PA, Kersten ML, Manegold S (2008) Breaking the memory wall in monetdb. Commun ACM 51(12):77–85
5. Chen Z, Gehrke J, Korn F (2001) Query optimization in compressed database systems. In: SIGMOD, pp 271–282. ACM: New York
6. Copeland GP, Khoshafian S (1985) A decomposition storage model. In: SIGMOD, pp 268–279. ACM: New York
7. Damme P (2017) Query processing based on compressed intermediates. VLDB PhD Workshop. Munich, 28.08.2017
8. Damme P, Habich D, Hildebrandt J, Lehner W (2017) Lightweight data compression algorithms: An experimental survey (experiments and analyses). In: EDBT, pp 72–83. Venice, 21–24.03.2017
9. Damme P, Habich D, Lehner W (2015) Direct transformation techniques for compressed data: General approach and application scenarios. In: ADBIS, pp 151–165. Springer
10. Damme P, Ungethüm A, Hildebrandt J, Habich D, Lehner W (2019) From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. ACM Trans Database Syst 44(3):9:1–9:46
11. Faerber F, Kemper A, Larson P, Levandoski JJ, Neumann T, Pavlo A (2017) Main memory database systems. Found Trends Databases 8(1-2):1–130
12. Habich D, Damme P, Ungethüm A, Lehner W (2018) Make larger vector register sizes new challenges?: Lessons learned from the area of vectorized lightweight compression algorithms. In: DBTest@SIGMOD, pp 8:1–8:6. ACM: New York
13. Habich D, Damme P, Ungethüm A, Pietrzyk J, Krause A, Hildebrandt J, Lehner W (2019) Morphstore – in-memory query processing based on morphing compressed intermediates LIVE. In: SIGMOD, pp 1917–1920. ACM: New York
14. He J, Zhang S, He B (2014) In-cache query co-processing on coupled CPU-GPU architectures. PVLDB 8(4):329–340
15. Hildebrandt J, Habich D, Damme P, Lehner W (2016) Compression-aware in-memory query processing: Vision, system design and beyond. In: ADMS, pp 40–56. Springer
16. Idreos S, Groffen F, Nes N, Manegold S, Mullender KS, Kersten ML (2012) Monetdb: two decades of research in column-oriented database architectures. IEEE Data Eng Bull 35(1):40–45
17. Karnagel T, Habich D, Lehner W (2017) Adaptive work placement for query processing on heterogeneous computing resources. PVLDB 10(7):733–744
18. Karnagel T, Müller R, Lohman GM (2015) Optimizing gpu-accelerated group-by and aggregation. In: ADMS, pp 13–24. Springer
19. Kissinger T, Schlegel B, Habich D, Lehner W (2013) QPPT: query processing on prefix trees. In: CIDR. Asilomar, 06.–09.01.2013
20. Komatsu K, Momose S, Isobe Y, Watanabe O, Musa A, Yokokawa M, Aoyama T, Sato M, Kobayashi H (2018) Performance evaluation of a vector supercomputer sx-aurora TSUBASA. In: SC, pp 54:1–54:12. IEEE/ACM: New York
21. Lang H, Kipf A, Passing L, Boncz PA, Neumann T, Kemper A (2018) Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In: DaMoN@SIGMOD, pp 5:1–5:8. ACM: New York
22. Lang H, Mühlbauer T, Funke F, Boncz PA, Neumann T, Kemper A (2016) Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: SIGMOD, pp 311–326. ACM: New York
23. Lee J et al (2014) Joins on encoded and partitioned data. PVLDB 7(13):1355–1366
24. Lemire D, Boytsov L (2015) Decoding billions of integers per second through vectorization. Softw Pract Exper 45(1):1–29
25. Li F, Das S, Syamala M, Narasayya VR (2016) Accelerating relational databases by leveraging remote memory and RDMA. In: SIGMOD, pp 355–370. ACM: New York
26. Li Y, Patel JM (2013) Bitweaving: Fast scans for main memory data processing. In: SIGMOD, pp 289–300. ACM: New York
27. Lisa NJ, Ungethüm A, Habich D, Lehner W, Nguyen TDA, Kumar A (2018) Column scan acceleration in hybrid CPU-FPGA systems. In: ADMS@VLDB, pp 22–33. Rio de Janeiro, 27.08.2018
28. Oukid I, Booss D, Lespinasse A, Lehner W, Willhalm T, Gomes G (2017) Memory management techniques for large-scale persistent-main-memory systems. PVLDB 10(11):1166–1177
29. Pietrzyk J, Habich D, Damme P, Lehner W (2019) First investigations of the vector supercomputer sx-aurora TSUBASA as a co-processor for database systems. In: BTW Workshopband, pp 33–50. GI: Bonn

Final edited form was published in "Datenbank-Spektrum". 19 (3), S. 183–197. ISSN: 1610-1995.
https://doi.org/10.1007/s13222-019-00323-w

30. Pietrzyk J, Ungethüm A, Habich D, Lehner W (2019) Fighting the duplicates in hashing: conflict detection-aware vectorization of linear probing. In: BTW, pp 35–53. GI: Bonn
31. Pirk H, Moll O, Zaharia M, Madden S (2016) Voodoo – A vector algebra for portable database performance on modern hardware. PVLDB 9(14):1707–1718
32. Polychroniou O, Raghavan A, Ross KA (2015) Rethinking SIMD vectorization for in-memory databases. In: SIGMOD, pp 1493–1508. ACM: New York
33. Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil EJ, O'Neil PE, Rasin A, Tran N, Zdonik SB (2005) C-store: a column-oriented DBMS. In: VLDB, pp 553–564. ACM: New York
34. Ungethüm A, Pietrzyk J, Damme P, Habich D, Lehner W (2018) Conflict detection-based run-length encoding – AVX-512 CD instruction set in action. In: ICDE Workshops, pp 96–101. IEEE Computer Society: Washington D.C.
35. Zukowski M, Héman S, Nes N, Boncz PA (2006) Super-scalar RAM-CPU cache compression. In: ICDE, p 59. IEEE Computer Society: Washington D.C.
36. Zukowski M, van de Wiel M, Boncz PA (2012) Vectorwise: a vectorized analytical DBMS. In: ICDE, pp 1349–1350. IEEE Computer Society: Washington D.C.