Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Nguyen Duy Anh Tuan, Akash Kumar

FPGA vs. SIMD: Comparison for Main Memory-Based Fast Column Scan

Erstveröffentlichung in / First published in:

Data Management Technologies and Applications: 7th International Conference. Porto, 26.-28.07.2018. Springer, S. 116-140. ISBN 978-3-030-26636-3.

DOI: http://dx.doi.org/10.1007/978-3-030-26636-3_6

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

FPGA vs. SIMD: Comparison for Main Memory-Based Fast Column Scan

Nusrat Jahan Lisa[1], Annett Ungethüm[1], Dirk Habich[1(✉)], Wolfgang Lehner[1], Nguyen Duy Anh Tuan[2], and Akash Kumar[2]

[1] Database Systems Group, Technische Universität Dresden, Dresden, Germany
{nusratjahan.lisa,annett.ungethum,dirk.habich, wolfgang.lehner}@tu-dresden.de
[2] Processor Design Group, Technische Universität Dresden, Dresden, Germany
{nguyenduyanh.tuan,akash.kumar}@tu-dresden.de

Abstract. The ever-increasing growth of data demands reliable database system with high-throughput and low-latency. Main memory-based column store database systems are state-of-the-art on this perspective, whereby data (values) in relational tables are organized by columns rather than by rows. In such systems, a full column scan is a fundamental key operation and thus, the optimization of the key operation is very crucial. This leads to have compact storage layout based fast column scan techniques through intra-value parallelism. For this reason, we investigated on different well-known fast column scan techniques using SIMD (Single Instruction Multiple Data) vectorization as well as using Field Programmable Gate Arrays (FPGA). Moreover, we present selective results of our exhaustive evaluation. Based on this evaluation, we find out the best column scan technique as per implementation mechanism– FPGA and SIMD. Finally, we conclude this paper via mentioning some lessons learned for our ongoing research activities.

Keywords: Column stores · Scan operation · Vectorization · FPGA · Pipeline

1 Introduction

The big data world challenging continuously on how to manage analytical complex database queries more efficiently along with high-throughput and low-latency. This leads to have fast database architecture. Therefore, to speedup the performance, database systems shifted from disk to main memory. Because storing as well as processing all data in main memory is faster than data stored on disk or on a flash drive [1,4]. Although it is important that all data must fit in main memory. However, main memories are still multi-gigabyte, while disk can be multi-terabyte. Additionally, organizing relational tables in main memory by column rather than by row effects the query performance positively as data stored in uniform pattern [1,20]. Thus, such database systems are called main

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

memory column store system. In order to increase the performance of analytical queries, two key aspects play an important role in this so-called main memory column store database systems. On the one hand, compressed storage layout is easily adaptable to reduce the amount of data [1,9,27]. On the other hand, main memory column stores are also smoothly adaptable for novel hardware features like vectorization using SIMD extensions [17,25], GPUs [8], FPGAs [19,22] or non-volatile main memory [16].

One of the primary operation in such system is *column scan* [7,13,23], because analytical queries usually compute aggregations over full or large parts of columns. Thus, the optimization of the scan primitive is very crucial [7,13,23]. Generally, the task of a column scan is to compare each entry of a given column against a given predicate and to return all matching entries. To efficiently realize this column scan, Lamport et al. [11] came up with the initial idea of intra-data processing on single processor word. Later, Li et al. [13] proposed a novel technique called *BitWeaving* which exploits the intra-instruction parallelism at the bit-level of modern processors. Intra-instruction parallelism means that multiple column entries are processed by a single instruction at once. In these approaches, multiple encoded column values are packed either horizontally or vertically into processor words providing high performance when fetching the entire column value [11,13]. Moreover, on these approaches query processing are happening directly on the packed processor words without unpacking the data items. Therefore, as the authors have shown, the more column values are packed in a processor word, the better scan performance [13].

Unfortunately, the length of processor words is currently fixed to 64-bit in common processors, which limits the performance of the intra-instruction parallelism based column scans. To overcome this limitation and to increase the intra-instruction parallelism, there exists two hardware-oriented opportunities. On the one hand, Single Instruction Multiple Data (SIMD) instruction set extensions such as Intel's SSE (Streaming SIMD Extensions) and 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 which reduce the instruction calls. The size of the vector registers ranges from 128 (Intel SSE 4.2) to 512-bit (Intel AVX-512), whereby these registers can be used instead of regular processor words. On the other hand, Field Programmable Gate Arrays (FPGAs) are an interesting alternative which provide a great deal of flexibility. Because it allows to design specialized configurable hardware components with arbitrary processor word sizes.

Our Contribution and Outline: This paper is the extended version of our paper [14], whereby we investigate both hardware-oriented opportunities using different types of column scan mechanism. Based on that, we make the following contributions.

1. In Sect. 2, we briefly recap the fast column scan techniques as foundation for our work.
2. The implementation using SIMD vector registers is discussed in Sect. 3, while Sect. 4 covers our FPGA implementation along with selective results of our

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

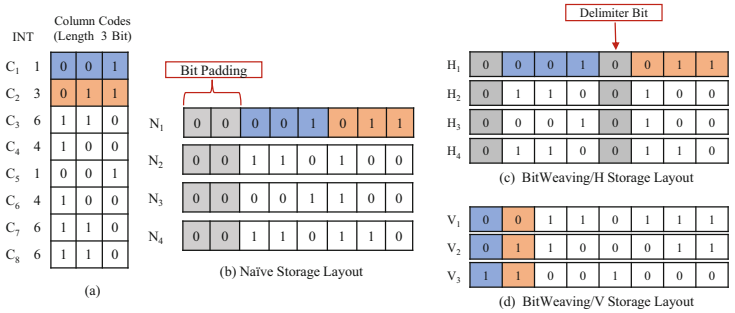

Fig. 1. Storage layout example with (a) 8 integer values with their 3-bit codes, (b) data representation in *Naïve* layout, (c) data representation in *BitWeaving/H* layout and (d) data representation in *BitWeaving/V* layout.

exhaustive evaluation. In particular, we separately evaluate each implementation followed by lessons learned summaries.

3. Finally, we conclude the paper with related work in Sect. 5 and a short conclusion in Sect. 6.

2 Column Scan

Column-oriented database management systems store relational data by columns rather than by rows [4,6]. The advantages are (i) that each column is separately considered and (ii) that the similarity of adjacent column values is preserved. Based on both advantages, the opportunity for compactness and the ability to process multiple column values at once is increased. Thus, the efficient realization of a column scan is an active research topic, whereby each approach consists of two components: (i) storage layout for column values and (ii) scan operation (predicate evaluation) on the proposed storage layout. In this paper, we compare two well-established approaches on two different hardware platforms.

2.1 Naïve

The first column scan technique is the Naïve approach, where each column is encoded with a fixed-length order-preserving code as illustrated in Fig. 1(a). The types of all column values including numeric and string types are encoded as unsigned integer codes [3,9]. The term *column code* refers to the encoded column values. To process multiple column codes in a single processor word during the scan operation, a intra-value parallelism-based compact storage layout is required. Lamport et al. [11] introduced a first approach for this. As shown in Fig. 1(b), column codes are continuously stored horizontally in processor words N_i. As stored codes are fixed in length, the extra unused bits in the processor word are padded with zeros. In our example, we use 8-bit processor words N_1 to N_4, such that two 3-bit column codes fit into one processor word including 2-bit padding per processor word.

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

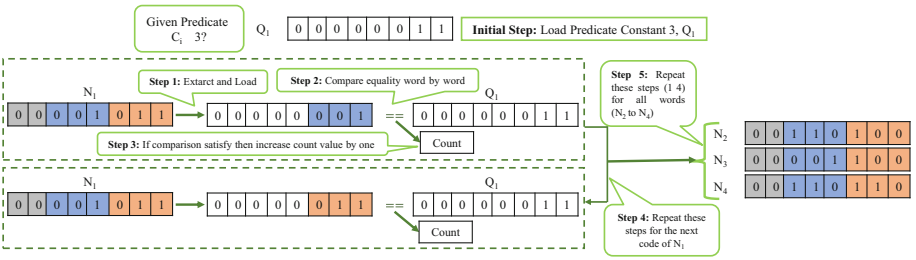

Fig. 2. Equality predicate evaluation using *Naïve/S* technique with extract-load-compare each column code.

During *Predicate Evaluation*, the task of a column scan is to compare each column code with a constant C and to output the number of *Count* indicating how many times the corresponding code satisfies the comparison condition. The predicate evaluation on *Naïve* layout can be done in two ways. *Firstly*, we can evaluate any predicate with simply extract, load and evaluate each (single) code with the comparison condition consecutively, without exploiting code-level parallelism. We named this technique as *Naïve/S*. Figure 2 describes the *equality* check in an exemplary way. The input from Fig. 1(b) is tested against the condition $C_i = 3$. The predicate evaluation steps are as follows:

Initially: Load the predicate constant *3* in word Q_1.
Step 1: Extract one code from N_1 and load in a temporary word.
Step 2: Check equality word-wise between Q_1 and temporary word.
Step 3: If comparison satisfies then increase the value of *Count* by one.
Step 4: Repeat Steps (1 to 3) for the next column code of N_1.
Step 5: Repeat Steps (1 to 4) for the rest of words N_2 to N_4.

Secondly, we can evaluate any predicate directly on the *Naïve* layout with exploiting code-level parallelism. The main advantage of such technique is predicate evaluation is done without decoupling the column codes from a word. We named this technique as *Naïve/M*. Figure 3 illustrated this technique in an exemplary way for the same input and test condition like *Naïve/S*. The detail steps are described as follows,

Initially: Load the *Naïve* layout of predicate constant *3* in Q_1.
Step 1: Check the equality bit-wise of each code between N_1 and Q_1 in parallel. There are 1-bit S_i flag registers for each code of *Naïve* word. For this example (Fig. 3), each word has two (S_1 and S_2) flag registers. If the condition satisfies then set one to S_i flags, otherwise set zero.
Step 2: Perform addition between S_1 and S_2, and store the result in *Count* word.
Step 3: Repeat Step 1 and Step 2 for the rest of words N_2 to N_4 in pipeline manner by overlapping instructions.

In both examples (Figs. 2 and 3), only the second code (C_2) satisfies the predicate, so the resulting *Count* value is one. In order to accelerate column scan,

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

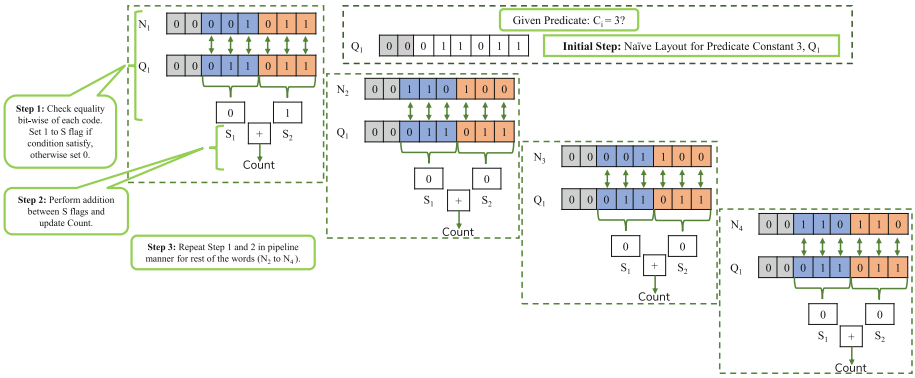

Fig. 3. Equality predicate evaluation using *Naïve/M* technique with directly evaluate on compact words.

Naïve/M technique is good choice than *Naïve/S* for two reason, i) *Naïve/M* technique evaluate predicate directly on the compact word, ii) it is using instruction overlapping mechanism which reduce the number of clock cycles significantly. However, *Naïve/M* technique is difficult to implement on common CPUs using 64-bit processor word, as common 64-bit word do not support intra-instruction parallelism.

2.2 BitWeaving

Our second considered column scan technique is called *BitWeaving* [13]. As illustrated in Fig. 1(a), *BitWeaving* also takes each column separately and encodes the column codes using a fixed-length order-preserving code (lightweight data compression [1,5]), whereby the types of all values including numeric and string types are encoded as an unsigned integer code [13]. To accelerate column scans, BitWeaving technique introduced two types of storage layouts along with an arithmetic framework instead of comparisons for predicate evaluations: *BitWeaving/H* and *BitWeaving/V* [13].

BitWeaving/H. In the storage layout of *BitWeaving/H*, the column codes of each column are viewed at the bit-level and the bits are aligned in memory in a way that enables the exploitation of the intra-cycle (intra-instruction) parallelism for the predicate evaluation. As illustrated in Fig. 1(c), column codes are continuously stored in processor words H_i, 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 H_1 to H_4, such that two 3-bit column codes fit into one processor word including one delimiter bit per code. The delimiter bit is used later to store the result of a predicate evaluation query.

To efficiently perform column scans using the *BitWeaving/H* storage layout, Li et al. [13] proposed an arithmetic framework to directly execute predicate

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

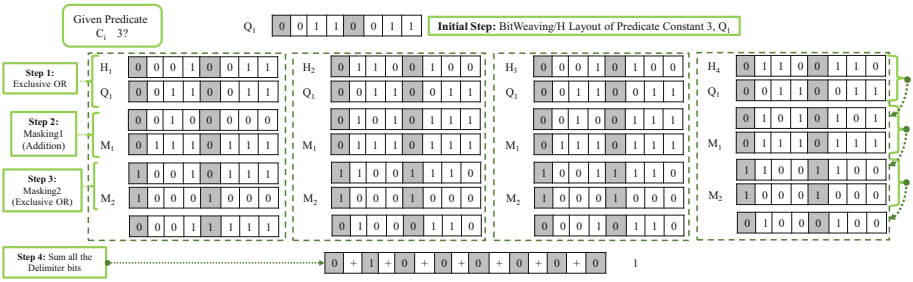

Fig. 4. Equality predicate evaluation using *BitWeaving/H* technique [13].

evaluations on the compressed data. 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-instruction parallelism) [13]. The supported predicate evaluations include equality, inequality, and range checks, whereby for each evaluation a function consisting of arithmetical and logical operations is defined [13]. Figure 4 highlights the *equality* check in an exemplary way. The input from Fig. 1(c) is tested against the condition $C_i = 3$. Then, the predicate evaluation steps are as follows:

Initially: Load the *BitWeaving/H* layout of predicate constant *3* in Q_1.
Step 1: Exclusive-OR operations between the words (H_1, H_2, H_3, H_4) and Q_1 are 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 is set to one and rest of all bits are set to zero) is performed.
Step 4: 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 the example of Fig. 4, only the second code (C_2) satisfies the predicate which is visible in the resulting bit vector.

BitWeaving/V. In *BitWeaving/V*, the codes are stored vertically across several processor words [13], such that one word contains one bit of several codes. Figure 1(d) shows how the column codes from Fig. 1(a) are stored in the *BitWeaving/V* layout. The words V_i are 8-bit long. The bits of the first number C_1 are stored at the first position of each word, the bits of the second number are stored at the second position, and so on. This way, eight 3-bit codes can be stored across three 8-bit words.

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

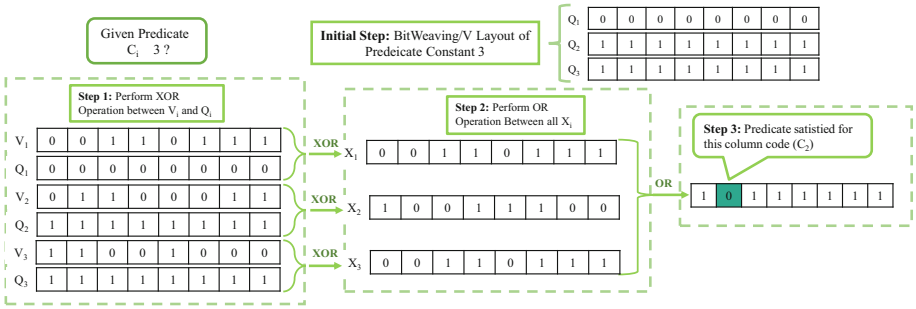

Fig. 5. Equality predicate evaluation using *BitWeaving/V* technique [13].

To evaluate predicate in this layout, we consider the restriction operation *Equality*. However, any kind of (restriction type) predicate evaluation can be perform in this layout. Figure 5 illustrated the *Equality* check predicate evaluation for *BitWeaving/V* in a exemplary way. In the example, we evaluate the column codes C_i for an equality with 3. The necessary steps are:

Initially: Predicate constant *3* is loaded as *BitWeaving/V* layout $(Q1, Q2, Q3)$.
Step 1: XOR operations are performed between *BitWeaving/V* layout based words and predicate constant as follows,

$$X_1 = V_1 \oplus Q_1$$
$$X_2 = V_2 \oplus Q_2$$
$$X_3 = V_3 \oplus Q_3$$

Step 2: Performed bitwise *OR* operations between (X_1, X_2, X_3).
Step 3: In the result word, there is only one position set to *0*. That means, the example condition is satisfied for only one column code and the total count value is one.

2.3 Summary

With the increasing demand for in-memory data processing, there is a critical need for fast scan operations [7, 13, 23]. The *Naïve/M* and *BitWeaving* techniques addresses this need by packing multiple codes into processor words and applying full-word instructions for predicate evaluations. As shown in [13], *BitWeaving* techniques achieved significant improvement over *Naïve/S* due to its intra-instruction parallelism mechanism. However, they do not consider *Naïve/M* technique. That defines explicitly the more codes are packed in processor words the better performance can be achieved. Unfortunately, processors words in all common CPUs are currently fixed to 64-bit in length. To further speedup *BitWeaving* or *Naïve/M*, larger processor words would be beneficial. To realize larger processor words, we have two hardware-oriented alternatives: (i) vector registers of SIMD extensions or (ii) Field Programmable Gate Arrays (FPGAs). We consider *BitWeaving* and *Naïve/M* approaches for both hardware-oriented alternatives. Both alternatives are discussed in the following sections in detail.

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

3 SIMD-Implementation

One hardware-based opportunity to optimize column scans is provided by vectorization using SIMD extensions (Single Instruction Multiple Data) of common CPUs. We developed SIMD-implementations for *Bitweaving/H*, *Bitweaving/V*, and *Naïve/M*. *Naïve/S* is intentionally left out because this equals a SIMD-Scan [23] when it is extended to SIMD. A comparison between a SIMD-Scan and the original *BitWeaving* variants has already been done by Li and Patel [13]. In the remainder of this chapter, we shortly introduce the system we used and its SIMD extensions. Then we explain our implementations in detail. Finally, all approaches are compared in an evaluation.

3.1 Target System

A SIMD implementation requires a system with the corresponding registers and instructions. Generally, SIMD instructions apply one operation to multiple elements of so-called vector registers at once. For a long time, the vector registers were 128-bit in size. 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 and Intel's AVX-512 uses even 512-bit for vector registers. The wider the vector registers, the more data elements can be stored and processed in a single vector. Additionally to an increased register size, each new vector extension comes with new instructions, e.g. gather-instructions were first introduced in AVX2. AVX-512 consists of several instruction sets, each providing different functionality, e.g. conflict detection or prefetching.

For the evaluation of our SIMD-implementation, we used an Intel Xeon Gold 6130 with DDR4-2666 memory offering SIMD extensions with vector registers of sizes 128-, 256-, and 512-bit (SSE, AVX2, and AVX-512). This system offers the AVX-512 Vector Length Extensions (VL), which provide most AVX-512 intrinsics for 128-bit and 256-bit registers, that would otherwise only work with 512-bit registers. There is a 32 KB L1 cache for instructions and 32 KB L1 for data. The L2 cache is 1 MB and the LLC (Last Level Cache) is 22 MB. The CPU runs at a base frequency of 2.1 GHz. It has 4 sockets, each containing 16 cores with up to two hyperthreads per core. However, the idea is to observe the influence of the different vector layouts and sizes, not the influence of multiple memory channels or CPU cores. Thus, all benchmarks are single threaded.

3.2 Implementation Details

The SIMD-implementation shows different challenges depending on the evaluation algorithm to be applied. For instance, *Naïve/M* could be implemented easily in regular registers. However, this would not be efficient because single bits cannot be addressed. This introduces an overhead to test whether a set of arbitrary bits, which may or may not be aligned within byte boundaries, is set

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

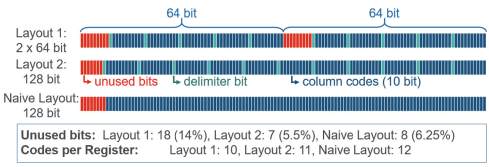

Fig. 6. Different variants to arrange column codes in a vector register.

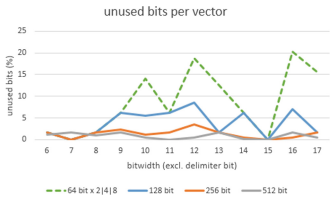

Fig. 7. Percentage of unused bits per vector register depending on the vector layout [14].

or not. A SIMD implementation has to solve this with a limited number of available instructions. Furthermore, while *BitWeaving/V* is trivially extended from the original approach to vector sizes, *BitWeaving/H* either has to make compromises in the usage of the registers, or work around the fact that the instruction set does not offer a full adder for numbers larger than 64-bit.

Naïve/M

Vector Storage Layout: The *Naïve* storage layout can easily be adapted to vector registers. Figure 6 shows different layouts in an exemplary way for 128-bit registers and 10-bit codes. The layouts for *BitWeaving/H* will be discussed in Sect. 3.2. The *Naïve* layout stores all codes consecutively in a register. Since codes are not spread across two registers, some bits of a register can remain unused. However, the *Naïve* layout is more compact than the *BitWeaving/H* layouts, because there are no delimiter bits.

Predicate Evaluation: The most simple predicate evaluation, which can directly be performed on data in the *Naïve* layout, is an equality check. For such an evaluation, two tasks have to be solved: (1) a bit-wise equality check between the input data and the predicate, and (2) a check for all code words in the input, if all bits of the comparison from step 1 are set. While task one can simply be done by applying an exclusive OR and negating the outcome, task two requires an additional bit-mask to filter the bits of each code word and perform the comparison. This is because we cannot explicitly access arbitrary bits of a vector register. The exact procedure for 128 bit is as follows:

1. Load the predicate in *Naïve* layout with `_mm_loadu_si128`
2. Load data in naïve layout (*input*) with `_mm_loadu_si128`
3. Perform bitwise XOR on the registers loaded in step 1 and step 2 with `_mm_xor_si128`
4. Negate the result from step 3. Perform a bit-wise AND with a vector, where the bits at the position of the current code are set to 1 and all other bits set to 0 (*filter*). `_mm_andnot_si128` performs both operations.
5. Compare the result from step 4 with *filter* using `_mm_cmpeq_epi32_mask`. The result is an 8-bit mask with the first four bits set to one if both vectors are equal.

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

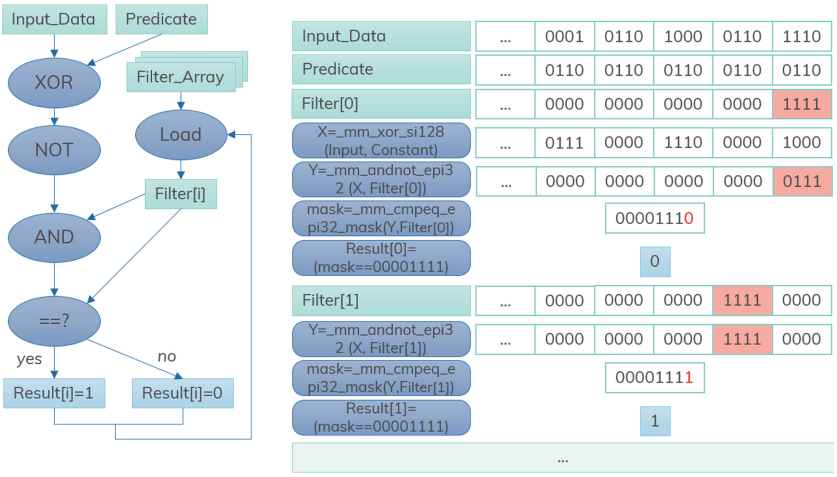

Fig. 8. Evaluation of data stored in the *Naïve* layout. The data is not extracted like in a SIMD-Scan but evaluated directly. The filter array is a precomputed array of vectors, where each vector acts as a filter to zero out every element except for one code word. On the left side, a graph shows the steps necessary for evaluating one input register. The right side shows this in an exemplary way for the first two code words of an input register. In this example, the bit-width of the codes is 4 and the vector width is 128 bit.

6. Compare the result from step 5 with an 8-bit number where the first four bits are set to one.
7. If all codes in *input* have been processed, repeat from step 2, else repeat from step 4 with the next code in *input*.

This procedure can be ported to 256 and 512 bit by simply renaming the intrinsics accordingly. For a better understanding, Fig. 8 illustrates the whole process for a single input register and provides an example for evaluating the first two codes in this register using SSE.

BitWeaving/H

Vector Storage Layouts: A straightforward way to implement *BitWeaving/H* using vector extensions is to load several 64-bit values containing the column codes and delimiter bits into a vector register. In this case, the original processor word approach is retained as proposed in *BitWeaving*. This vector layout is shown as *Layout 1* in Fig. 6. However, this method does not use the register size optimally. For instance, in a 128-bit register, there is space for 11 column codes with a bit width of 10 and their delimiter bits (see Fig. 6 *Layout 2*), but *Layout 1* can only hold 10 codes. In *Layout 2*, we treat the vector register as a full processor word and arrange the column codes according to the vector register size. Figure 7 shows the percentage of unused register space for different register

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

sizes and both layouts, where the dashed line shows the usage for *Layout 1* and the remaining lines for *Layout 2*. As we can see, *Layout 2* makes better use of the vector register. For our evaluation in Sect. 3.3, we implemented both layouts.

Predicate Evaluation: Like in the original approach, the query evaluation on data in the *BitWeaving/H* layout in vector registers consists of a number of bit-wise operations and one addition. The exact bit-wise operations and their sequence depends on the comparison operator. For instance, a *smaller than* comparison or an *equality* check requires XOR operations and an addition as shown in Sect. 2.2. For counting the number of results quickly, an AND is also necessary. For 512-bit registers, this is realized by using AVX-512 intrinsics. The following steps are necessary for a *smaller than* comparison if the data is using the vector *Layout 1* (see Fig. 6):

1. The predicate and the data in *BitWeaving/H* layout is loaded with `_mm512_loadu_si512`. The constraint must only be loaded once.
2. The bit-wise XOR is performed with `_mm512_xor_si512`.
3. The addition is performed with `_mm512_add_epi64`.
4. Optional: To set only the delimiter bits, an AND between the precomputed inverted bit-mask and the result from step 3 is performed with `_mm512_and_si512`.
5. Optional: For counting the number of set delimiter bits `_mm512_popcnt_epi64` is applied.
6. Optional: The result from step 5 can be further reduced by adding the individual counts with `_mm512_reduce_add_epi64`.
7. Finally, the result is stored with `_mm512_storeu_si512`. If only the number of results is required, this step can be skipped. Afterwards, a new iteration starts at step 1.

Note that the SIMD intrinsics for step 5 and 6 do not exist for 128-bit and 256-bit registers. In these cases, the result is written back to memory and treated conventionally, i.e. like an array of 64-bit values.

These steps work for *Layout 1* but not for *Layout 2*. This is because in step 3, a full adder is required. However, this functionality is supported for words containing 16, 32, or 64 bits, but not for 128, 256, or 512 bits. Hence, this adder must be implemented by the software.

Full Adder for Large Numbers: While *Layout 2* uses the size of the vector register more efficiently, it comes with a major drawback: There is no full adder for more than 64 bit on recent CPUs. The evaluation with *BitWeaving/H* uses mainly bit-wise operations but one addition is necessary in all evaluations, i.e. equality, greater than, and smaller than. To realize this addition for 128-, 256-, or 512-bit, there are two different ways: (a) the addition is done by iterating through the bits of the summands and determining and adding the carry bit in every step, and (b) only the carry at the 64-bit boundaries is determined and added to the subsequent 64-bit value. Option (a) requires sequential processing and cannot

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

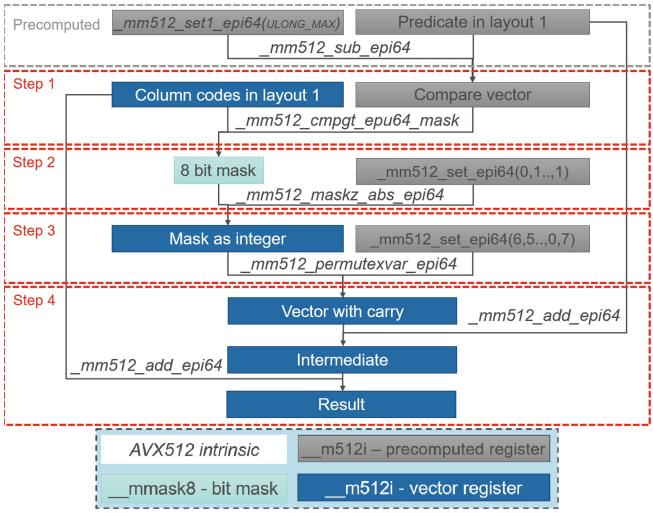

Fig. 9. A software adder for large numbers using AVX-512 intrinsics. For *BitWeaving*, the two summands are the predicate and the column codes. This approach can easily be adapted for 128 and 256 bits [14].

be implemented in a vectorized way. Thus, we chose option (b). The exact steps for option 2 are shown in Fig. 9 for 512-bit vector registers:

1. Since the result of the addition of two 64-bit values is also 64-bit, a potential overflow cannot be determined directly. Instead, we subtract one summand from the largest representable number and check whether the result is larger than the other summand. If it is smaller, there is a carry. This can be done vectorized. The output of the comparison between two vector registers containing unsigned 64-bit integers is a bit-mask.
2. The bit-mask resulting from step 1 is used on a vector containing only the decimal number 1 as 64-bit value at every position.
3. A carry is always added to the subsequent 64-bit value. For this reason, the result from step 2 is shifted to the left by 64-bit. This is realized by intrinsics providing a permutation of 64-bit values.
4. Finally, the two summands and the result from step 3 are added.

All steps can be done using AVX-512 intrinsics. If one of the summands is a constant, like the predicate in *BitWeaving*, the subtraction in step 1 can be precomputed.

BitWeaving/V. The implementation of vectorized *BitWeaving/V* is straightforward because all needed functionality is provided by the SIMD intrinsics of SSE, AVX2, and AVX-512. The layout stays exactly the same as described in Sect. 2.2. In our case, the processor words V_i are 128-bit, 256-bit, or 512-bit long. The number of necessary words for a segment equals the number of bits

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

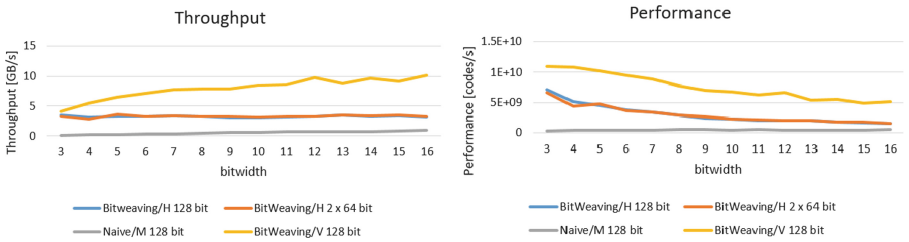

Fig. 10. Throughput and performance of all presented 128-bit implementations for growing code sizes.

per code word. The evaluation is also done as described in Sect. 3.2. For instance, an equality check using AVX-512 requires the following steps for each segment:

1. Load the first word of the segment and a vector filled with the 1st bit of the predicate with `_mm512_loadu_si512`.
2. Perform a bit-wise XOR on the registers loaded in step 1 with `_mm512_xor_si512`
3. Invert result from step 1. Perform a bit-wise AND with a 1-vector if it is the first word of the segment, perform bit-wise AND with the result from the last iteration otherwise. The inverting and bit-wise AND are done with `_mm512_andnot_si512`.
4. Repeat from step one with next word of the segment and the next bit of the predicate.

3.3 Evaluation and Summary

In the evaluation, we want to observe the influence of the different vector layouts and sizes, not the influence of multiple memory channels or CPU cores. Thus, all benchmarks are single threaded. All measurement values are averaged over ten runs.

Overview. Figure 10 shows a comparison of the performance (codes/s) and the throughput (GB/s) of all implementations using 128 bit. As expected, the *Naïve* layout provides the lowest throughput and performance. The time needed for evaluating every code in a register individually cannot make up for the slightly better usage of the available bits. The two layouts of *BitWeaving/H* do not show any significant differences but perform better than the *Naïve/M* approach.

Finally, *BitWeaving/V* shows the highest throughput and performance as could be expected since it is the approach with the least operations, which have to be performed while the input vector layout is more compact than in *BitWeaving/H*. Moreover, it has the smallest output size, resulting in less store operations, i.e. the bits containing the evaluation result are stored consecutively in the result register. *BitWeaving/V* is the only implementation, where the throughput increases when the code size increases, while *Naïve/M* and *BitWeaving/H* show

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

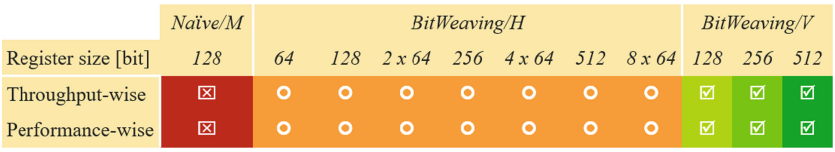

Fig. 11. Comparison of the implemented column scan techniques.

a constant throughput. A reason for this behaviour is that in *BitWeaving/V*, the number of result bits per input bit decreases when the code size increases because more data is needed to compute a result. This leads to less store operations for the same amount of input data. For instance, if the bit width of the codes is 3, one segment consists of 3 processor words. Thus, the result, i.e. one processor word, is written back after these 3 processor words have been evaluated. But if the bit width is 15, there are 15 processor words, which are evaluated before one processor word is written back to memory. At the same time, the performance decreases for all *BitWeaving* approaches while the size of the codes increases. In *BitWeaving/H*, this is because less codes fit into one processor word when the code size increases. In *BitWeaving/V*, it takes more operations before a result is computed as explained before. Before going into detail, Fig. 11 shows an overview of all implementations. While *BitWeaving/V* stays clearly on top of the other approaches, it also shows some variation between the different vector sizes. *BitWeaving/H* is less influenced by the vector size.

BitWeaving/H. For codes containing 3 bits and a delimiter bit, the non-optimized 64 bit implementation achieves a throughput of 2.9 GB/s, which equals a performance of almost 58e+8 codes per second.

The results for 3-bit column codes for all different horizontal vector layouts are shown in Table 1. All values are averaged over 10 runs. The results show, that there is a performance gain when using the vectorized approach, but it is not as significant as expected. For instance, we would expect a 100% speed-up when changing from 64 to 128 bits since we can process twice the data at once. Unfortunately, the throughput and the performance increase only by 14%. Moreover, it even decreases when changing from 256 to 512 bits for both vector layouts. However, these numbers can only provide a rough estimation since the throughput varies by up to 0.5 GB/s between the individual runs.

Figure 12 shows the throughput and performance for all implemented *BitWeaving/H* versions and different code bitwidths. For comparison, we also implemented a scalar 64-bit *BitWeaving/H* version without any further optimization for special cases, such that the predicate evaluation is always executed in the same way.

The differences between the vectorized implementations and the scalar implementation becomes even smaller when the code size increases while the throughput oscillates between 2.5 GB/s and 4 GB/s for all versions (see Fig. 12). There is a mere tendency of the 256-bit implementations to provide the best performance

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

Table 1. Evaluation results on Intel Xeon Gold 6130, 3 bits per code, average over 10 runs [14].

Vector layout	Throughput (GB/s)	Performance (Codes/s)
None (64-bit) (baseline)	2.9	57.8e+8
2X64-bit (Layout 1)	3.3	65.7e+8
4X64-bit (Layout 1)	3.5	69.3e+8
8X64-bit (Layout 1)	2.9	57.6e+8
128-bit (Layout 2)	3.6	71.6e+8
256-bit (Layout 2)	3.6	72.4e+8
512-bit (Layout 2)	2.9	58.9e+8

in average and for the 512-bit versions to provide the least performance. Nevertheless, the insignificance of the differences cannot be explained with the query evaluation itself. To find the bottleneck, we deleted the evaluation completely, such that only the vectorized load and store instructions were left. Then, we measured the throughput again and received results between 3 GB/s and 4 GB/s. A simple `memcopy` had a stable performance around 4.5 GB/s. Hence, in contrast to the naive implementation, the vectorized implementations are bound by the performance of loading and storing data, while the peak throughput cannot become larger than 4.5 GB/s.

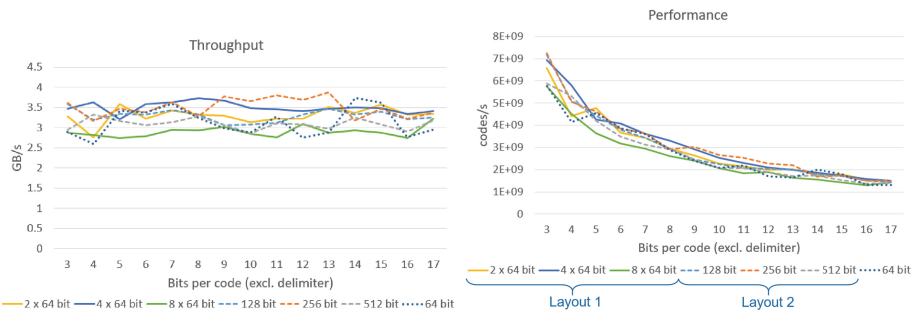

Fig. 12. Throughput and performance for *BitWeaving/H* [14].

BitWeaving/V. The performance and throughput of all implemented *BitWeaving/V* versions for different code sizes are shown in Fig. 13. Contrary to *BitWeaving/H*, there is a clear increase of performance and throughput when the register size increases. A reason for this is the already mentioned smaller output size. Unlike in *BitWeaving/V*, in the horizontal approach, there is a padding between the result bits, which is as wide as a code word. To get these result bits,

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

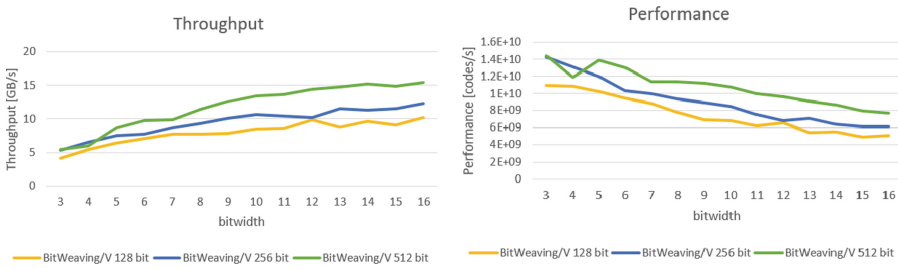

Fig. 13. Throughput and performance for *BitWeaving/V*.

the whole vector word has to be extracted to several regular registers, where the bits can be shifted together, or even written back to memory completely if there are not enough registers. Since it is common for CPU cores to have only 16 general purpose registers, this worst-case is the usual case. However, *BitWeaving/V* does not have such a padding, which makes the output more compact and reduces store operations. This relaxes the memory bandwidth bottleneck to a certain degree. This is especially obvious in the throughput for larger code sizes, where there are more input registers processed before the output register is written back. The performance decrease for 512 bit at a code size of 4 bit is reproducible. It comes with a throughput, which is not increased as much as expected. We did not find an explanation for this in the algorithm itself, especially because it only occurs for 512 bit. A possible reason is a fail of the optimizer during compilation. To test this theory, we compiled the exactly same source code with icc, whereas we were using gcc before. The results did not show the decrease at 4 bit. Instead, there is a peak at 10 bit and the overall increase is less steady. Thus, it is safe to assume that these outliers are caused by the compiler rather than the implementation or the hardware.

4 FPGA-Implementation

Besides the implementation by means of wider vector registers, the second hardware based implementation possibility is the use of Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits, which are re-configurable after being manufactured. More specifically, a hardware description language, e.g., Verilog, is used to describe the hardware modules. This description is then translated via several steps to an implementation for the FPGAs. From the perspective of intra-value or intra-instruction parallelism based storage layout, the advantage of FPGAs is that we are able to use an arbitrary length of processor word in the custom hardware design.

4.1 Target System

Modern FPGAs are integrated with MPSoC (multiprocessor system on chip) architectures. The Xilinx® Zynq UltraScale+™ platform—our target FPGA

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

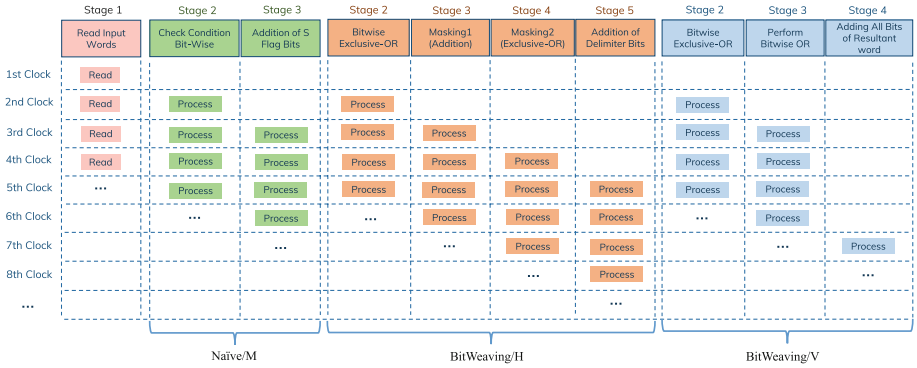

Fig. 14. Pipeline-based PE for different intra-instruction parallelism based column scan techniques.

system—is such a system containing not only Programmable Logic (PL) based FPGA, but also has MPSoC-based Processing System (PS) in particular having four ARM® Cortex-A53 cores with 32 KB of L1 instruction cache resp. 32 KB data cache per core and a 1MB shared L2 cache. The main memory consists of two memory modules (DDR4-2133) with the accumulated capacity of 4.5 GB. Although the main memory of our targeted FPGA platform has limitations regarding capacity and bandwidth compared to modern Intel systems, the flexibility to prepare any type of custom hardware and the high parallelism criteria of FPGAs are very beneficial to overcome these challenges.

4.2 Implementation Detail

Inside the PL area of FPGAs, we can develop Processing Element (PE) modules for any type of predicates using Configurable Logic Block (CLB) slices, where each CLB slice consists of Look-up Tables (LUTs), Flip-Flops (FFs), and cascading adders [22]. As illustrated in Fig. 14, the stages of PEs are processing words in pipeline manner through overlapping instructions, whereby we developed 3-stage, 5-stage and 4-stage pipeline-based PE for equality check predicate evaluation on the basis of *Naïve/M*, *BitWeaving/H* and *BitWeaving/V* techniques as introduced in Figs. 3, 4 and 5, respectively. All PEs have a common *Stage 1* of reading data words from main memory (see Fig. 14). Rest in every stages a specific task is performed as shown in Fig. 14, whereby the stages for different techniques are grouped by colors. The detail of *Naïve/M* pipeline stages are:

Stage 2: Check equality condition bit-wise and set S flag values according to the condition satisfying result,
Stage 3: Perform addition between S flags in order to count the matched column codes.

Then, the detail of *BitWeaving/H* pipeline stages are:

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

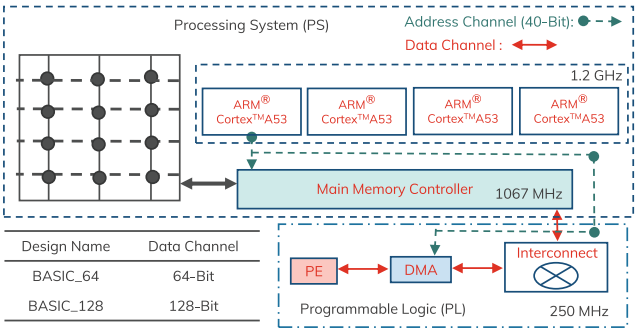

Fig. 15. Basic architecture [14].

Stage 2: Executing bit-wise Exclusive-OR operations,
Stage 3: Masking operations (Addition),
Stage 4: Masking operations (Exclusive-OR) using predefined mask registers to prepare the output word,
Stage 5: Adding delimiter bits of output words in order to count the matched column codes.

Finally, the detail of *BitWeaving/V* based pipelines are:

Stage 2: Executing bit-wise Exclusive-OR operations,
Stage 3: Executing bit-wise OR operations,
Stage 4: Adding all bits of previous stage resultant words in order to count the matched codes (this stage would execute after every w cycles, whereas w is the width of column code).

For all cases, we write only the final output word of count to the main memory. This is not shown in Fig. 14 as it is a non-pipeline stage which executes once only. Therefore, the total number of cycles for *Naïve/M*, *BitWeaving/H* and *BitWeaving/V* is $(n+3)$, $(n+5)$ and $(n+4)$, respectively, where n is the total number of input words.

Basic Architecture. We started with developing 64-bit word based hardware design as Basic architecture (BASIC_64) and subsequently increased the word width to 128-bit (BASIC_128) (see Fig. 15). In this architecture, we use Direct Memory Access (DMA) between the main memory and the PE, in order to reduce the load of the ARM core and to reduce the latency of accessing the main memory. We prepared basic architecture based designs having either *Naïve/M* or *BitWeaving/H* or *BitWeaving/V* technique based PE, whereas each design is processing either 64-bit or 128-bit words.

Hybrid Architecture. The main challenge comes up when the word to be processed become larger than 128-bit, because the width of the data channel

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

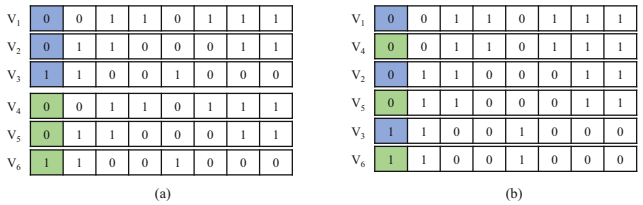

Fig. 16. *BitWeaving/V* storage layout patterns, (a) for basic and (b) for hybrid architectures.

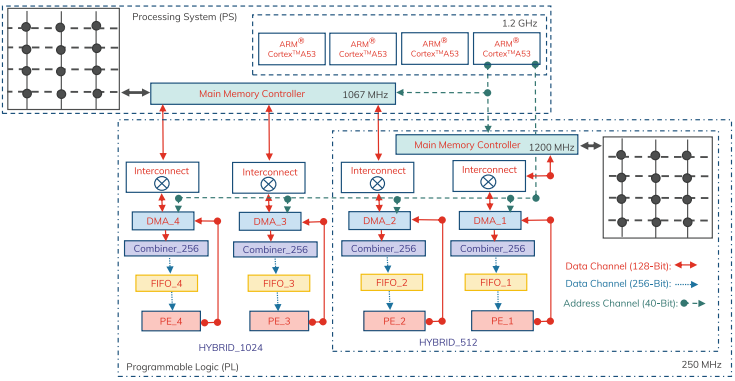

Fig. 17. Hybrid architecture [14].

of the main memory can only be extended up to 128-bit although the PEs are capable to handle word sizes beyond 128-bit. To tackle this challenge, we developed a hybrid architecture based on multiple DMAs, where each DMA is accessing the main memory via an independent data channel. As a consequence, we replicate our PE and DMA a few times depending on the number of available main memory data channels.

Moreover, two main memory modules are available on our targeted FPGA platform as mentioned earlier: one is connected with the PS and the other one is connected to the PL. The PS part main memory has four data channels, while the PL part has only one. However, maximum channel width is 128-bit. So, maximum five times of 128-bit words can be processed in parallel by using multiple main memory modules. However, having maximum number of data channels in a design saturates the bandwidth of main memory. Therefore, we can prepare another custom hardware module, whereas 128-bit words can be combined into larger words. Thus, we implemented and replicated a custom combiner (namely Combiner_256) to combine two 128-bit words to produce 256-bit word. This introduces another stage in each proposed pipeline design, such that each PE is processing a 256-bit word in each clock cycle. Such a combiner can easily adoptable in *Naïve/M* and *BitWeaving/H* techniques based hardware designs as they stored codes in words horizontally rather than vertically like *BitWeav-*

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

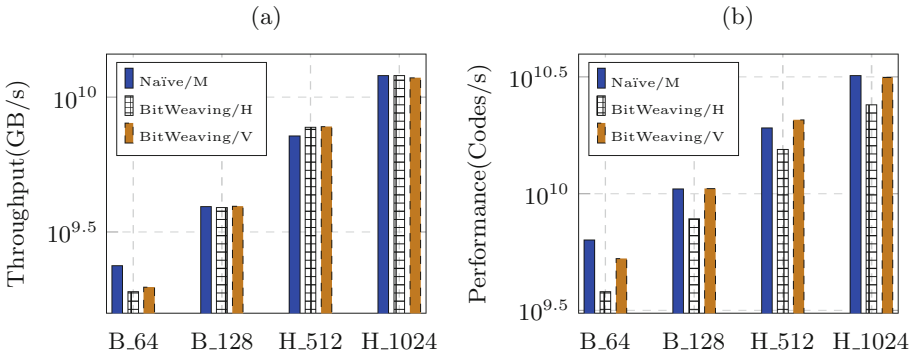

Fig. 18. Analysis on (a) Throughput-wise, (b) Performance-wise for basic and hybrid architectures using different column scan techniques (3-Bit Per Code).

ing/V. Therefore, for *BitWeaving/V*, the input words are stored alternatively rather than sequentially as illustrated in Fig. 16(b) for 3-bit column codes, so that combiner can merge two words perfectly without breaking the sequence of codes. However, we keep the as usual storage pattern of *BitWeaving/V* for basic architecture as described in Sect. 2.2 (see Fig. 16(a)).

In addition, we use appropriate depth based FIFO between the combiners and the PEs to synchronize IO transmission between PEs and main memory, whereas main memory is using stream-based data transmission. This avoids an overflow of the DMA buffer. By mixing all the above mentioned concepts, we prepared hybrid architecture based designs as HYBRID_512 and HYBRID_1024, to process two and four times of 256-bit word in parallel in order to make 512-bit and 1024-bit words, respectively for all techniques (see Fig. 17).

4.3 Evaluation and Summary

Experiments are evaluated using two main metrics: throughput (GB/s) and performance (Codes/s). Although in our previous work, we evaluated energy consumption metric as estimated energy and actual energy for codes per joule on *BitWeaving/H* scan [14]. But in these evaluations, we did not consider energy consumption due to having same behaviour like performance as it depends on codes. In addition, we showed that, our proposed basic and hybrid architectures win over ARM-based evaluations for *BitWeaving/H* scan [14]. Therefore, these evaluations are targeted to analysis the behaviour between *Naïve/M*, *BitWeaving/H*, *BitWeaving/V* column scan techniques for basic and hybrid architectures. We evaluated with BASIC_64, BASIC_128, HYBRID_512 and HYBRID_1024 designs for *Naïve/M*, *BitWeaving/H* and *BitWeaving/V* scan techniques, whereby Fig. 18 shows the results for 3-bit column codes (excluding delimiter bit for *BitWeaving/H* scan) with equality check predicate.

We started with BASIC_64 design based evaluations and found that, *Naïve/M* provides higher throughput than *BitWeaving* techniques (see Fig. 18(a)).

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

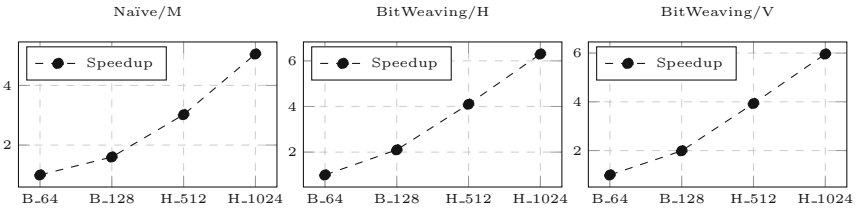

Fig. 19. Analysis in terms of Speedup between basic and hybrid architectures for all column scan techniques.

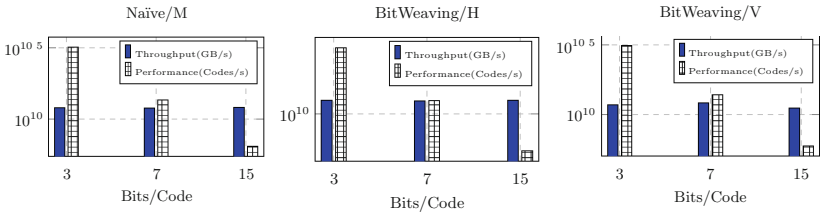

Fig. 20. Analysis on HYBRID_1024 design using different column scan techniques for different number of bits per code.

Because it able to execute on 300 MHz frequency due to having simple logic instruction based technique, whereas others execute on 250 MHz. However, this scenario changed for BASIC_128, HYBRID_512 and HYBRID_1024 based designs, where we achieved approximately same throughput for all techniques (see Fig. 18(a)) as the frequency of these designs are identical. Moreover, different number of total clock cycles of PEs for different techniques as shown in Sect. 4.2 do not effect the throughput due to its pipeline mechanism. In the hybrid architectures-based designs data words are uniformly distributed among the PEs. In addition, the hybrid architecture based designs are processing beyond 256-bit width based data words through multiple main memory data channels and also flexible to use additional hardware (i.e., Combiner_256, FIFO), which is not available on BASIC_64 and BASIC_128 designs. As a consequence, for all techniques, HYBRID_1024 gives the peak throughput of approx. 12 GB/s, whereas three data channels from PS part main memory and one data channel from PL part main memory are used. Although the PS part main memory have maximum four data channels. But using the maximum number of channels in parallel saturates the bandwidth of PS part main memory. So, in HYBRID_1024 we used multiple main memories in order to have four individual data channels.

Performance-wise evaluation varies between different techniques. *BitWeaving/H* provides always less performance in terms of codes per second among all techniques (see Fig. 18(b)). In BASIC_64 design, *Naïve/M* provides the highest performance (see Fig. 18(b)). However, rest in all designs the performance become marginal between *Naïve/M* and *BitWeaving/V* (see Fig. 18(b)). There are two reasons. On the one side, the number of *bit padding* increases in *Naïve/M*

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

Table 2. Resource utilization for HYBRID_1024 designs.

Scan tech	LUTs (%)	FFs (%)
Naïve/M	12.89	8.64
BitWeaving/H	13.68	9.5
BitWeaving/V	13.99	9.15

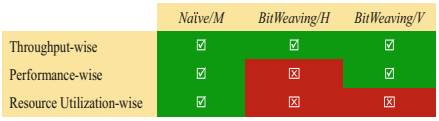

Fig. 21. Evaluation matrix-wise analysis on the column scan techniques.

technique based BASIC_128, HYBRID_512 and HYBRID_1024 designs exponentially than BASIC_64 as the word size increases. As mentioned earlier, hybrid architecture merged two 128-bit words to make one 256-bit word. So, there are 2-bit *bit padding* in one 128-bit word for 3-bit column code. It extend to 4-bit *bit padding* for 256-bit word and so on. As a consequence, we are losing number of codes per word as the word size increases which effects the performance. On the other side, there is no chance of losing codes in *BitWeaving/V* as each bit of a code is store vertically per word (see Fig. 1(d)). This makes the marginal balance of processing codes per second between *Naïve/M* and *BitWeaving/V*. Therefore, performance-wise *Naïve/M* and *BitWeaving/V* both win over *BitWeaving/H*.

Technique-wise the behavior of throughput and performance are identical among the basic and hybrid architectures (see Fig. 18). Therefore, the speedup for main memory based intra-value parallelism based scan techniques among the basic and hybrid architectures on the targeted FPGA platform is linear (see Fig. 19), whereas the BASIC_64 design is the baseline. This defines, that the HYBRID_1024 design is best for all mentioned column scan techniques on FPGAs.

We also evaluated different numbers of bits per (column) code for three mentioned techniques using the best design: HYBRID_1024 (see Fig. 20). In this case symmetrical behavior found between all techniques, whereby a linearly decreasing behavior found for performance as the bits per code increases except the throughput. The reason is— increasing the code size decreases the number of codes per word which negatively effects the performance which is evaluated on the basis of the number of codes as expected, whereas throughput evaluation is independent of codes.

Table 2 illustrated the overall resource utilization in terms of LUTs (%) and FFs (%) for the best design HYBRID_1024 among all techniques using Xilinx® resource analyzer, whereby *Naïve/M* technique requires most optimum resource than the others due to its straight-forward predicate evaluation mechanism. After all kind of evaluations we found that, throughput-wise all techniques showed identical behaviour, performance-wise *Naïve/M* and *BitWeaving/V* techniques are better than *BitWeaving/H*, but resource utilization-wise *Naïve/M* technique is the most optimum one. Finally, these leads us to conclude that, *Naïve/M* technique is the best technique for FPGAs (see Fig. 21).

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

5 Related Work

Generally, the efficient utilization of SIMD 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 [5,12,24]. On the other hand, SIMD instructions are also used in other database operations like scans [7,23], aggregations [25] or joins [2].

Most research in the direction of FPGA optimization focused on creating custom hardware modules for different types of database query operations [10, 15,18,22,26]. For example, *Ziener et al.* presented concepts and implementations for hardware acceleration for almost all important operators appearing in SQL queries [26]. Moreover, *Sidler et al.* explored the benefits of specializing operators for the Intel Xeon+FPGA machine, where the FPGA has coherent access to the main memory through the QPI bus [18]. *Teubner et al.* performed XML projection on FPGAs and report on performance improvements of several factors [21].

Ever-increasing amount of data leads the recent research on main memory column store database system, whereas column stores are more effective performance-wise than row stores. In addition, it allows to evaluate query directly on the intra-data parallelism based compact storage layout. For that, there are several research has happened on how to efficiently evaluate query directly on the compact storage layout in order to improve the column scan performance, whereas the scan is one of the most important primitives in main memory database systems [7,13]. But to the best of our knowledge, none of the existing works investigated, firstly the domain of FPGA-accelerated data scan, secondly the comparison behavior as per intra-data parallelism based column scan mechanisms between FPGA-based and SIMD-based hardware implementation.

6 Conclusions

A key primitive in main memory column store database systems is *column scan* [7,13,23], because analytical queries usually compute aggregations over full or large parts of columns. Thus, the optimization of the scan primitive is very crucial [7,13,23]. In this paper, we explored two hardware-based implementation opportunities for scan optimization using SIMD extensions and custom architectures on FPGA on different scan mechanisms. In particular, we analysis the behavioral differences between *Naïve* [11] and *BitWeaving* [13] scan mechanisms as per hardware-based implementation. With both implementation, we are able to improve the scan performance, whereas the FPGA is best for *Naïve* technique and *BitWeaving* is perfect for SIMD. Therefore, to improve scan performance through FPGA do not require any fancy scan mechanism as *BitWeaving* due to its high parallelism criteria and flexibility to configure hardware as per requirements.

Final edited form was published in "Data Management Technologies and Applications: 7th International Conference. Porto 2018", S. 116-140. ISBN: 978-3-030-26636-3
https://doi.org/10.1007/978-3-030-26636-3_6

References

1. Abadi, D.J., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the SIGMOD, pp. 671–682 (2006)
2. Balkesen, C., Alonso, G., Teubner, J., Özsu, M.T.: Multi-core, main-memory joins: sort vs. hash revisited. PVLDB **7**(1), 85–96 (2013)
3. Binnig, C., Hildenbrand, S., Färber, F.: Dictionary-based order-preserving string compression for main memory column stores. In: Proceedings of the SIGMOD, pp. 283–296 (2009)
4. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in monetdb. Commun. ACM **51**(12), 77–85 (2008)
5. Damme, P., Habich, D., Hildebrandt, J., Lehner, W.: Lightweight data compression algorithms: an experimental survey (experiments and analyses). In: Proceedings of the EDBT, pp. 72–83 (2017)
6. Faerber, F., Kemper, A., Larson, P., Levandoski, J.J., Neumann, T., Pavlo, A.: Main memory database systems. Found. Trends Databases **8**(1–2), 1–130 (2017)
7. Feng, Z., Lo, E., Kao, B., Xu, W.: ByteSlice: pushing the envelop of main memory data processing with a new storage layout. In: Proceedings of the SIGMOD, pp. 31–46 (2015)
8. He, J., Zhang, S., He, B.: In-cache query co-processing on coupled CPU-GPU architectures. PVLDB **8**(4), 329–340 (2014)
9. Hildebrandt, J., Habich, D., Damme, P., Lehner, W.: Compression-aware in-memory query processing: vision, system design and beyond. In: ADMS Workshop at VLDB, pp. 40–56 (2016)
10. István, Z., Sidler, D., Alonso, G.: Caribou: Intelligent distributed storage. PVLDB **10**(11), 1202–1213 (2017)
11. Lamport, L.: Multiple byte processing with full-word instructions. Commun. ACM **18**(8), 471–475 (1975)
12. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. Softw. Pract. Exp. **45**(1), 1–29 (2015)
13. Li, Y., Patel, J.M.: BitWeaving: fast scans for main memory data processing. In: Proceedings of the SIGMOD, pp. 289–300 (2013)
14. Lisa, N.J., Ungethüm, A., Habich, D., Nguyen, T.D.A., Kumar, A., Lehner, W.: Column scan optimization by increasing intra-instruction parallelism. In: Proceedings of the DATA, pp. 344–353. SciTePress, Setúbal (2018)
15. Mueller, R., Teubner, J., Alonso, G.: Data processing on FPGAs. Proc. VLDB Endow. **2**(1), 910–921 (2009). 10.14778/1687627.1687730
16. Oukid, I., Booss, D., Lespinasse, A., Lehner, W., Willhalm, T., Gomes, G.: Memory management techniques for large-scale persistent-main-memory systems. PVLDB **10**(11), 1166–1177 (2017)
17. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: Proceedings of the SIMD, pp. 1493–1508 (2015)
18. Sidler, D., István, Z., Owaida, M., Alonso, G.: Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In: Proceedings of the SIGMOD, pp. 403–415 (2017)
19. Sidler, D., Istvan, Z., Owaida, M., Kara, K., Alonso, G.: doppioDB: a hardware accelerated database. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD 2017, pp. 1659–1662. ACM, New York (2017). https://doi.org/10.1145/3035918.3058746. http://doi.acm.org/10.1145/3035918.3058746

20. Stonebraker, M., et al.: C-store: a column-oriented DBMS. In: Proceedings of the VLDB, pp. 553–564 (2005)
21. Teubner, J.: FPGAs for data processing: current state. IT Inf. Technol. **59**(3), 125–131 (2017). https://doi.org/10.1515/itit-2016-0046
22. Teubner, J., Woods, L.: Data Processing on FPGAs. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, San Rafael (2013)
23. Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., Schaffner, J.: SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. VLDB **2**(1), 385–394 (2009)
24. Zhao, W.X., Zhang, X., Lemire, D., Shan, D., Nie, J., Yan, H., Wen, J.: A general SIMD-based approach to accelerating compression algorithms. ACM Trans. Inf. Syst. **33**(3), 1–28 (2015)
25. Zhou, J., Ross, K.A.: Implementing database operations using SIMD instructions. In: Proceedings of the SIGMOD, pp. 145–156 (2002)
26. Ziener, D., Bauer, F., Becher, A., Dennl, C., Meyer-Wegener, K., Schürfeld, U., et al.: FPGA-based dynamically reconfigurable SQL query processing. ACM Trans. Reconfig. Technol. Syst. **9**(4), 25:1–25:24 (2016)
27. Zukowski, M., Héman, S., Nes, N., Boncz, P.A.: Super-scalar RAM-CPU cache compression. In: Proceedings of the ICDE, p. 59 (2006)