Column Scan Optimization by Increasing Intra-Instruction Parallelism

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

[1]*Database Systems Group, Technische Universität Dresden, Dresden, Germany*

[2]*Processor Design Group, Technische Universität Dresden, Dresden, Germany*

{*jahan_lisa.nusrat, annett.ungethuem, dirk.habich,tuan_duy_anh.nguyen, akash.kumar, wolfgang.lehner*}*@tu-dresden.de*

Keywords: Column Stores, Scan Operation, Vectorization, FPGA, Pipeline.

Abstract: The key objective of database systems is to reliably manage data, whereby high query throughput and low query latency are core requirements. To satisfy these requirements for analytical query workloads, in-memory column store database systems are state-of-the-art. In these systems, relational tables are organized by column rather than by row, so that a full column scan is a fundamental key operation and thus, the optimization of the key operation is very crucial. For this reason, we investigated the optimization of a well-known scan technique using SIMD (Single Instruction Multiple Data) vectorization as well as using Field Programmable Gate Arrays (FPGA). In this paper, we present both optimization approaches with the goal to increase the intra-instruction execution parallelism to process more columns values in a single instruction simultaneously. For both, we present selective results of our exhaustive evaluation. Based on this evaluation, we draw some lessons learned for our ongoing research activities.

1 INTRODUCTION

Processing of complex analytical database queries with low-latency and high throughput on an ever-increasing amount of data is a major challenge in our data-driven world. To tackle that challenge, the database architecture shifted from a disk-oriented to a main memory-oriented approach (Abadi et al., 2006; Boncz et al., 2008) due to the availability of high main memory capacities on modern hardware systems. 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 (Stonebraker et al., 2005; Boncz et al., 2008) 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 (Stonebraker et al., 2005; Boncz et al., 2008). To further increase the performance of analytical queries, two key aspects play an important role in these so-called column store database systems. On the one hand, data compression is used to reduce the amount of data (Abadi et al., 2006; Hildebrandt et al., 2016; Zukowski et al., 2006). On the other hand, in-memory column stores constantly adapt to novel hardware features like vectorization using SIMD extensions (Polychroniou et al., 2015; Zhou and Ross, 2002), GPUs (He et al., 2014), FPGAs (Sidler et al., 2017b; Teubner and Woods, 2013a) or non-volatile main memory (Oukid et al., 2017).

A key primitive in these systems is a *column scan* (Feng et al., 2015; Li and Patel, 2013; Willhalm et al., 2009), because analytical queries usually compute aggregations over full or large parts of columns. Thus, the optimization of the scan primitive is very crucial (Feng et al., 2015; Li and Patel, 2013; Willhalm et al., 2009). 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, Li et al. (Li and Patel, 2013) 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 their horizontal approach, multiple compressed column values are packed horizontally into processor words providing high performance when fetching the entire column value (Li and Patel, 2013). As the authors have shown, the more column values are packed in a processor word, the better the BitWeaving scan performance (Li and Patel, 2013).

Unfortunately, the length of processor words is currently fixed to 64-bit in common processors, which

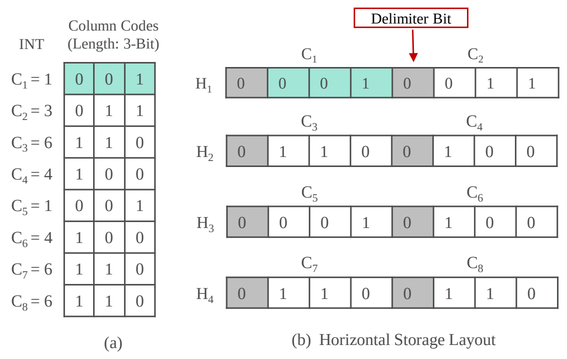


Figure 1: BitWeaving example with (a) 8 integer values with their 3-bit codes and (b) data representation in *BitWeaving/H* layout.

limits the performance of the BitWeaving scan. To overcome this limitation and to increase the intra-instruction parallelism for BitWeaving, there exists two hardware-oriented opportunities. On the one hand, Single Instruction Multiple Data (SIMD) instruction set extensions such as Intels 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. The available operations include parallel arithmetic, logical, and shift operations as well as permutations, whereby the size of the vector registers ranges from 128 (Intel SSE 4.2) to 512-bit (Intel AVX-512). These vector registers can be used instead of regular processor words. On the other hand, Field Programmable Gate Arrays (FPGAs) are an interesting alternative which allows to design specialized hardware components with arbitrary processor word sizes.

Our Contribution and Outline: In this paper, we investigate both optimization opportunities. Based on that, we make the following contributions.

1. In Section 2, we briefly recap the BitWeaving technique as foundation for our work.
2. The optimization using SIMD vector registers is discussed in Section 3, while Section 4 covers our FPGA optimization.
3. In Section 5, we present selective results of our exhaustive evaluation. In particular, we separately evaluate each optimization, followed by a global lessons learned summary.

Finally, we conclude the paper with related work in Section 6 and a short conclusion in Section 7.

2 BITWEAVING

Generally, the *BitWeaving* technique is aligned to columnar data and can be used as native storage organization technique in state-of-the-art in-memory column store database systems (Li and Patel, 2013). As illustrated in Figure 1(a), BitWeaving takes each column separately and encodes the column values using a fixed-length order-preserving code (lightweight data compression (Abadi et al., 2006; Damme et al., 2017)), whereby the types of all values including numeric and string types are encoded as an unsigned integer code (Li and Patel, 2013). The term *column code* refers to the encoded column values. To accelerate column scans in such cases, BitWeaving consists of a storage layout and an arithmetic framework for predicate evaluations (Li and Patel, 2013).

2.1 Storage Layout

In the storage layout, 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 circuit-level intra-cycle (intra-instruction) parallelism for the predicate evaluation. For that, BitWeaving comes with two storage variants, a horizontal version, *BitWeaving/H*, and a vertical version, *BitWeaving/V* (Li and Patel, 2013). In this paper, we mainly focus on *BitWeaving/H*, because many lightweight data compression algorithms are designed for such a horizontal layout (Abadi et al., 2006; Damme et al., 2017). To convert horizontal compressed column codes to a vertical BitWeaving layout, additional work is required.

As illustrated in Figure 1(b), column codes are continuously stored in processor words 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 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.

2.2 Predicate Evaluation

Baed on that, 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 column scans using the *BitWeaving/H* storage layout, Li et al. (Li and Patel, 2013) proposed an arithmetic framework to directly execute predicate evaluations on the compressed data. There

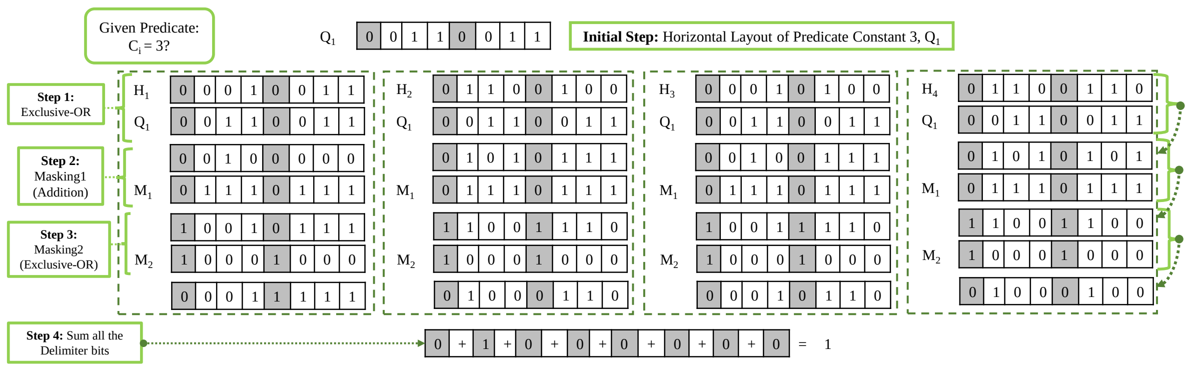

Given Predicate: $C_i = 3$?
Q_1
Initial Step: Horizontal Layout of Predicate Constant 3, Q_1
Step 1: Exclusive-OR
Step 2: Masking1 (Addition)
Step 3: Masking2 (Exclusive-OR)
Step 4: Sum all the Delimiter bits
H_1
H_2
H_3
H_4
M_1
M_2

Figure 2: Equality predicate evaluation with the *BitWeaving/H* technique (Li and Patel, 2013).

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) (Li and Patel, 2013). The supported predicate evaluations include equality, inequality, and range checks, whereby for each evaluation a function consisting of arithmetical and logical operations is defined (Li and Patel, 2013).

Figure 2 highlights the *equality* check in an exemplary way. The input from Figure 1(b) 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 the words (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 is set to one and rest of all bits are 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 Figure 2, only the second code (C_2) satisfies the predicate which is visible in the resulting bit vector.

2.3 Summary

With the increasing demand for in-memory data processing, there is a critical need for fast scan operations (Feng et al., 2015; Li and Patel, 2013; Willhalm et al., 2009). The *BitWeaving* approach addresses this need by packing multiple codes into processor words and applying full-word instructions for predicate evaluations using a well-defined arithmetic framework. As shown in (Li and Patel, 2013), the more codes that can be packed into a processor word, the more codes can be processed in parallel leading to better performance (intra-instruction parallelism). Unfortunately, processors words in all common CPUs are currently fixed to 64-bit in length. To further speedup BitWeaving, there are two interesting options. First, this can be achieved using better compression techniques, which however is algorithmically limited and highly dependent on the data characteristics. Second, 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). Both optimization alternatives are discussed in the following sections in detail.

3 SIMD-OPTIMIZATION

One hardware-based opportunity to optimize the BitWeaving scan is provided by vectorization using SIMD extensions (Single Instruction Multiple Data) of common CPUs. 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.

3.1 Vector Storage Layouts

A naïve 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 Figure 3. The evaluation works exactly as described in section 2, but instead of arithmetic operators, the corresponding SIMD instructions are used to process 2, 4, or 8 64-bit values at once. 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 Figure 3 *Layout 2*), but *Layout 1* can only hold 10 codes. In *Layout 2*, we treat the vector register as full processor word and arrange the column codes according to the vector register size. Figure 4 shows the percentage of unused register space for different register 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 Section 5, we implemented both layouts.

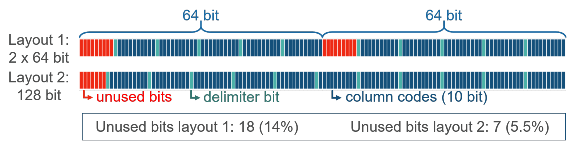


Figure 3: Different variants to arrange column codes in a vector register.

3.2 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 bitwise operations and one addition. The exact bitwise 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 Section 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 Figure 3):

1. The query constraint and the data in a BitWeaving/H layout is loaded with `_mm512_loadu_si512`. The constraint must only be loaded once.
2. The bitwise 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 bitmask 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.

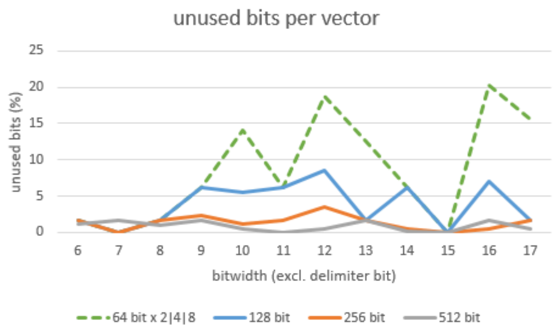


Figure 4: Percentage of unused bits per vector register depending on the vector layout.

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.

3.3 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 bitwise 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 be implemented in a vectorized way. Thus, we chose option (b). The exact steps for option 2 are shown in Figure 5 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 bitmask.
2. The bitmask 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.

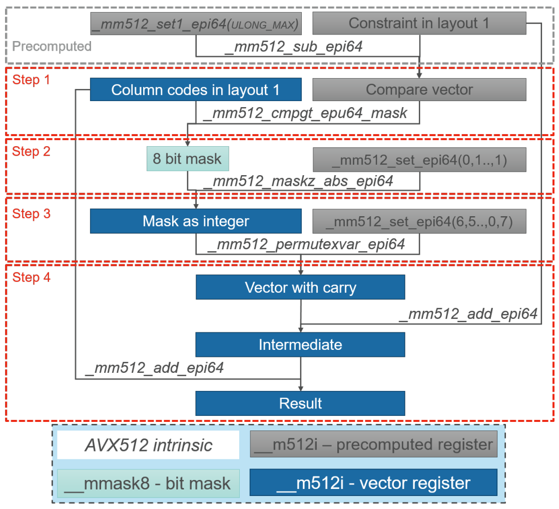


Figure 5: A software adder for large numbers using AVX-512 intrinsics. For BitWeaving, the two summands are the constraint and the column codes. This approach can easily be adapted for 128 and 256 bits.

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

3.4 Summary

As described above, the original BitWeaving/H approach can be ported to SIMD extensions in various ways. In Section 5, we will evaluate all possibilities in detail.

4 FPGA-OPTIMIZATION

Besides the optimization by means of wider vector registers, the second hardware-based optimization possibility is the use of Field Programmable Gate Arrays (FPGAs). FPGAs are integrated circuits, which are configurable after being manufactured. More specifically, a hardware description language, e.g., Verilog, is used to describe a custom hardware module. This description is then translated via several steps to an implementation for the FPGAs. From the perspective of BitWeaving/H, the advantage of FPGAs is that we are able to use an arbitrary length of processor word in our custom made BitWeaving/H hardware.

4.1 Target FPGA System

Modern FPGAs are based on MPSoC (multiprocessor system on chip) architectures. The Xilinx® Zynq UltraScale+™ platform—our target FPGA system—is such an MPSoC-based FPGA board containing not only programmable logic but also 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 (Xilinx, 2017). The main memory consists of two memory modules (DDR4-2133) with the accumulated capacity of 4.5GB. 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 as described in the following.

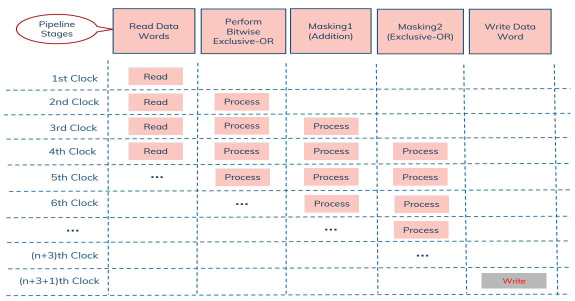


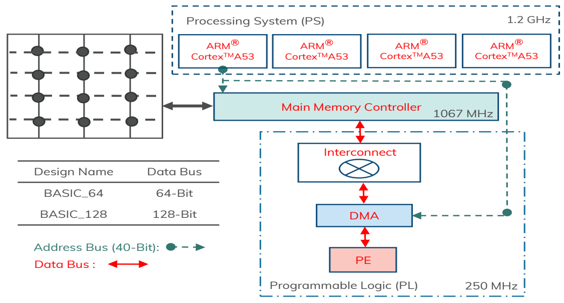
Figure 6: Pipeline-based PE for BitWeaving/H scan.

4.2 Basic Architecture

Inside the Programmable Logic (PL) area of FPGAs, we can develop Processing Elements (PE) 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 (Teubner and Woods, 2013a). For BitWeaving/H, we developed a 5-stage pipeline-based PE for equality check predicate evaluation on the basis of *Layout 2* as introduced in the previous section. In this case, a specific task is performed in each stage of the pipeline as shown in Figure 2:

Stage 1: reading data words from main memory,
Stage 2: executing bitwise 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: finally writing the output word to the main memory.

As illustrated in Figure 6, all stages are processed in parallel (pipeline parallelism). In order to reduce the load of the ARM cores and to reduce the latency of accessing the main memory, we use Direct Memory Access (DMA) between the main memory and our PE implementing BitWeaving/H. We started with 64-bit width based data words as a basic architecture (BASIC_64) and subsequently increased the word width to 128-bit (BASIC_128) (see Figure 7). That means, we are reading either 64-bit or 128-bit data words in each cycle and these data words are processed as proposed in BitWeaving/H.



Design Name	Data Bus
BASIC_64	64-Bit
BASIC_128	128-Bit

Figure 7: Basic Architecture.

4.3 Hybrid Architecture

The main challenge comes up when the words to be processed become larger than 128-bit, because the width of the data channel 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 channels, while the PL part has only one. However, each channel can transmit a maximum of 128-bit data words. Therefore, five times of 128-bit words can be processed in parallel by using multiple main memory modules. Using additional custom hardware, these 128-bit words can be combined into larger words, which can then be processed in parallel. 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 into the pipeline design (see Figure 6), such that each PE is processing a 256-bit word in every clock cycle. In addition, we use a FIFO between the combiners and the PEs to decouple the input pool of the PEs from the stream based data transmission between the main memory and the DMA. This avoids an overflow of the DMA buffer.

Using all the mentioned concepts, we prepared new designs, namely HYBRID_512 and HYBRID_1024, to process two and four times of 256-bit width based words in parallel in order to allow for 512-bit and 1024-bit width based data words, respectively (see Figure 8).

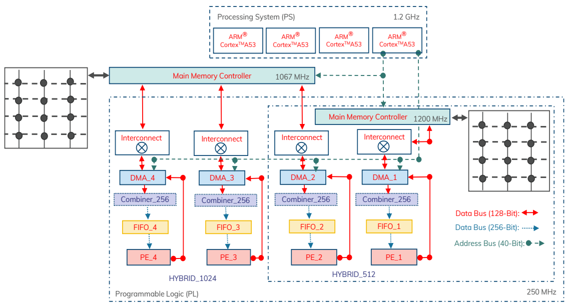


Figure 8: Hybrid Architecture.

5 EVALUATION

This section contains the evaluation results of our presented optimization techniques, whereby we separately evaluate each optimization. Afterwards, we draw some lessons learned from these evaluations. In addition to a performance analysis, we also look at the energy behavior, because energy is more and more a limiting factor.

5.1 SIMD-Optimization

The evaluation of our SIMD-optimization was done on an Intel Xeon Gold 6130 with DDR4-2666 memory offering SIMD extensions with vector registers of sizes 128-, 256- and 512-bit. The idea is to observe the influence of the different vector layouts and sizes,

Table 1: Evaluation Results on Intel Xeon Gold 6130, 3 Bits Per Code, Average over 10 Runs.

Vector Layout	Throughput(GB/s)	Performance(Codes/s)	Energy(Codes/J)
none (64-bit) (baseline)	2.9	57.8e+8	29.2e+6
2X64-bit (Layout 1)	3.3	65.7e+8	32e+6
4X64-bit (Layout 1)	3.5	69.3e+8	34.7e+6
8X64-bit (Layout 1)	2.9	57.6e+8	29.4e+6
128-bit (Layout 2)	3.6	71.6e+8	35.4e+6
256-bit (Layout 2)	3.6	72.4e+8	36e+6
512-bit (Layout 2)	2.9	58.9e+8	29.3e+6

Table 2: Evaluation Results for ARM-based, Basic and Hybrid Execution (3 Bits Per Code).

Hardware Design Name	Throughput (GB/s)	Performance (Codes/s)	Est. Energy (Codes/J)	Act. Energy (Codes/J)
ARM_64 (64-bit words, Single Core)	1.9	38e+8	N/A	18e+7
ARM_256 (4X64-bit words, Quad Core)	6	120e+8	N/A	55e+7
BASIC_64 (64-bit words, Basic Arch.)	1.9	38e+8	10e+8	18e+7
BASIC_128 (128-bit words, Basic Arch.)	3.9	78e+8	20e+8	35e+7
HYBRID_512 (2X256-bit words, Hybrid Arch.)	7.7	155e+8	31e+8	67e+7
HYBRID_1024 (4X256-bit words, Hybrid Arch.)	12	240e+8	48e+8	105e+7

not the influence of multiple memory channels or CPU cores. Thus, all benchmarks are single threaded. For comparison, we also implemented a naive 64-bit BitWeaving/H version without any further optimization for special cases, such that the predicate evaluation is always executed in the same way. For codes containing 3 bits and a delimiter bit, this naive implementation achieves a throughput of 2.9 GB/s, which equals a performance of almost 58e+8 codes per second. Additionally, we retrieved the energy consumption of all 4 power domains on the CPU and summed them up. For this first test case with 3-bit codes and a naive implementation (baseline), 29.2e+6 codes can be processed with every Joule.

The results for 3-bit column codes for all different 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 9 shows the performance for all column code sizes between 3 and 17 bits, while Figure 11 shows the energy efficiency. The differences between the vectorized implementations and the naive implementation becomes even smaller when the code

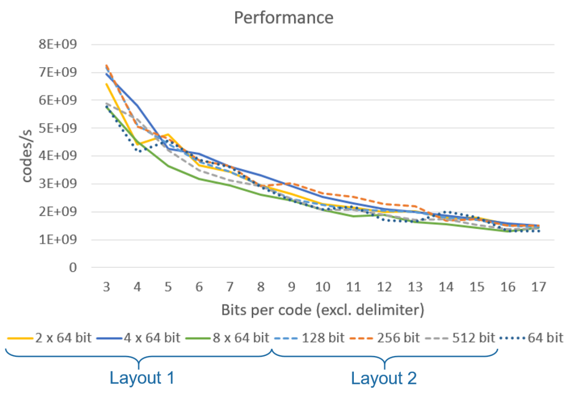


Figure 9: Performance (SIMD-Optimization).

size increases while the throughput oscillates between 2.5 GB/s and 4 GB/s for all versions (see Figure 10). There is a mere tendency of the 256-bit implementations to provide the best performance 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. The same applies for energy efficiency. That means, the SIMD-optimization does not achieve the desired result neither from performance nor from energy perspective.

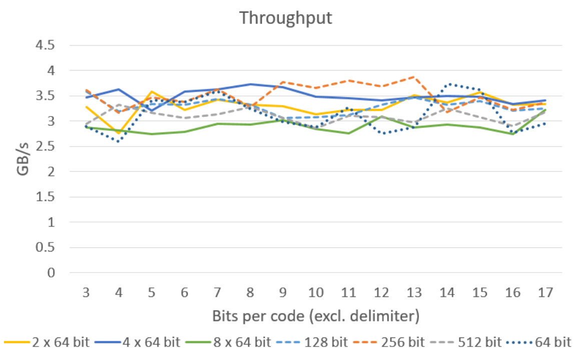


Figure 10: Throughput (SIMD-Optimization).

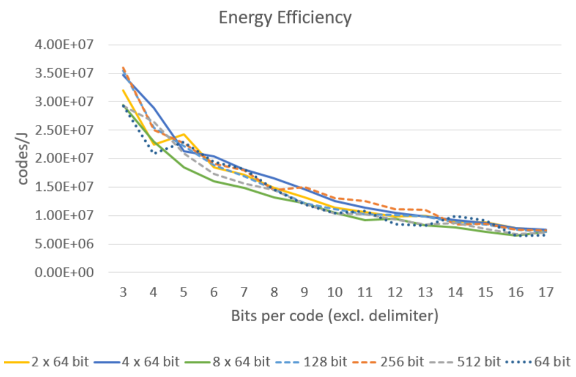


Figure 11: Energy Efficiency (SIMD-Optimization).

5.2 FPGA-Optimization

As done in the previous section, all designs are evaluated using three metrics: throughput (GB/s), performance (Codes/s) and energy (Codes/J). Two different ways are used for energy consumption measurement. Firstly, we measured energy consumption using Xilinx® Power Tool as Estimated Energy. Secondly, we used external power meter device as Actual Energy. Actual Energy in terms of Codes/J is less than Estimated Energy, because the Xilinx® Power Tool estimate power consumption only for a specific custom design, whereas the power meter device measure the real time power consumption for the entire FPGA board.

In order to prove the efficiency of our basic and hybrid architectures for BitWeaving/H scan, we prepared also ARM-based implementations as baseline. Therefore, we implemented single and quad ARM cores based designs namely ARM_64 and ARM_256, respectively, where each core is processing 64-bit width based data words as proposed by the original BitWeaving/H approach. We evaluated these two approaches, whereby Table 2 shows the results for 3-bit column codes (excluding delimiter bit) with equality check predicate during the BitWeaving/H scan. The data words are uniformly distributed among the cores for ARM_256 design. In addition, only Actual Energy is measured for ARM-based designs because these designs are not implemented using the Xilinx® tool.

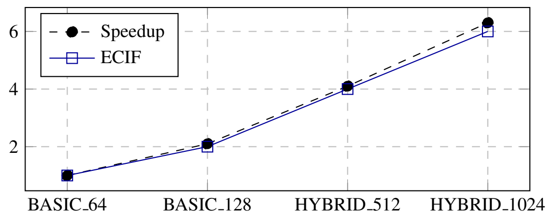


Figure 12: Analysis in terms of Speedup and Energy Consumption Improvement Factor (ECIF) between Basic and Hybrid Architectures.

Afterwards, we evaluated our basic and hybrid architectures-based designs for 3-bit column codes (excluding delimiter bit) with equality check predicate during the BitWeaving/H scan (see Table 2), where data words are uniformly distributed among the PEs for hybrid designs. As we can see, the ARM_64 and BASIC_64 gives the same performance, throughput and actual energy, because in each clock cycle both of them are processing the same width based data words which is 64-bit and both are accessing PS part main memory via one data channel. ARM_256 is better than BASIC_128 for all evaluated metrics because it is processing four times of 64-bit width based data words in parallel, whereas BASIC_128 is processing two times of 64-bit. On the other side, 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 and not possible on ARM_64 and ARM_256 designs due to its non-customized hardware architecture. As a consequence, HYBRID_1024 gives the peak throughput of 12GB/s, whereas three data channels from PS part main memory and one data channel from PL part main memory are used. The behavior of all evaluated metrics are identical among the basic and hybrid designs (see Table 2). Therefore, the speedup and the energy consumption improvement factor for main memory-based BitWeaving/H scan among the basic and hybrid designs on the targeted FPGA platform is linear (see Figure 12), whereas the BASIC_64 design is the baseline. This may lead to the conclusion, that the hybrid architecture-based designs are very beneficial for BitWeaving/H scan on FPGAs.

Finally, we evaluated different numbers of bits per (column) code using the best design: HYBRID_1024

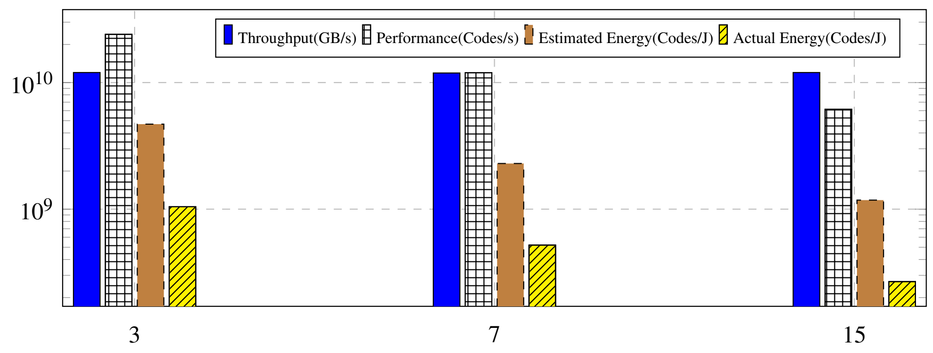


Figure 13: Analysis on HYBRID_1024 for Different Bits Per Code.

(see Figure 13). A linearly decreasing behavior found among all metrics as the bits per code increases except the throughput. The reason is, that the increase of bits per code decreases the number of codes per data word which negatively effects on those evaluated metrics which are evaluated on the basis of the number of codes as expected, whereas throughput evaluation is independent of codes.

5.3 Lesson Learned

The hardware-based optimization of BitWeaving/H by means of SIMD or FPGA is possible. However, the FPGA optimization is superior to SIMD optimization from a performance as well as an energy perspective. Although the performance can be slightly increased with SIMD optimization, it can not be increased as much as expected. Furthermore, higher vector widths bring no further advantages, since the main memory bandwidth is already fully utilized for low vector register sizes. Thus, 128-bit vector registers with a BitWeaving/H vector storage *Layout 2* deliver the best performance on our test hardware. In contrast to SIMD, the FPGA optimization brings a significant increase in performance with a high energy efficiency. In this case, a data width of 1024-bit for BitWeaving/H delivers the best performance. Thus, FPGA optimization should be well-investigated for database systems in the near future.

6 RELATED WORK

Generally, the efficient utilization of SIMD instructions in database systems is a very active research field (Polychroniou et al., 2015; Zhou and Ross, 2002). On the one hand, these instructions are frequently applied in lightweight data compression algorithms (Damme et al., 2017; Lemire and Boytsov, 2015; Zhao et al., 2015). On the other hand, SIMD instructions are also used in other database operations like scans (Feng et al., 2015; Willhalm et al., 2009), aggregations (Zhou and Ross, 2002) or joins (Balkesen et al., 2013).

Most research in the direction of FPGA optimization focused on creating custom hardware modules for different types of database query operations up to now (Sidler et al., 2017a; István et al., 2017; Mueller et al., 2009; Teubner and Woods, 2013b; Ziener et al., 2016). For example, *Ziener et al.* presented concepts and implementations for hardware acceleration for almost all important operators appearing in SQL queries (Ziener et al., 2016). 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 (Sidler et al., 2017a). They focused on two commonly used SQL operators for strings: LIKE, and REGEXP_LIKE, and provide a novel and efficient implementation of these operators in reconfigurable hardware. *Teubner et al.* performed XML projection on FPGAs and report on performance improvements of several factors (Teubner, 2017).

To the best of our knowledge, none of the existing works investigated the domain of FPGA-accelerated data scan, whereby the scan is one of the most important primitives in in-memory database systems.

7 CONCLUSIONS

A key primitive in in-memory column store database systems is a *column scan* (Feng et al., 2015; Li and Patel, 2013; Willhalm et al., 2009), because analytical queries usually compute aggregations over full or large parts of columns. Thus, the optimization of the scan primitive is very crucial (Feng et al., 2015; Li and Patel, 2013; Willhalm et al., 2009). In this paper, we evaluated two hardware-based optimization opportunities using SIMD extensions and custom architectures on FPGA for the BitWeaving scan technique (Li and Patel, 2013). With both optimizations, we are able to improve the scan performance, whereas the FPGA optimization is superior to SIMD optimization from a performance and energy perspective.

REFERENCES

Abadi, D. J., Madden, S., and Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682.

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

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

Damme, P., Habich, D., Hildebrandt, J., and Lehner, W. (2017). Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83.

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

He, J., Zhang, S., and He, B. (2014). In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB*, 8(4):329–340.

Hildebrandt, J., Habich, D., Damme, P., and Lehner, W. (2016). Compression-aware in-memory query processing: Vision, system design and beyond. In *ADMS Workshop at VLDB*, pages 40–56.

István, Z., Sidler, D., and Alonso, G. (2017). Caribou: Intelligent distributed storage. *PVLDB*, 10(11):1202–1213.

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

Li, Y. and Patel, J. M. (2013). Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300.

Mueller, R., Teubner, J., and Alonso, G. (2009). Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921.

Oukid, I., Booss, D., Lespinasse, A., Lehner, W., Willhalm, T., and Gomes, G. (2017). Memory management techniques for large-scale persistent-main-memory systems. *PVLDB*, 10(11):1166–1177.

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

Sidler, D., István, Z., Owaida, M., and Alonso, G. (2017a). Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD*, pages 403–415.

Sidler, D., Istvan, Z., Owaida, M., Kara, K., and Alonso, G. (2017b). doppiodb: A hardware accelerated database. In *SIGMOD*, pages 1659–1662.

Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E. J., O'Neil, P. E., Rasin, A., Tran, N., and Zdonik, S. B. (2005). C-store: A column-oriented DBMS. In *VLDB*, pages 553–564.

Teubner, J. (2017). Fpgas for data processing: Current state. *it - Information Technology*, 59(3):125.

Teubner, J. and Woods, L. (2013a). *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Teubner, J. and Woods, L. (2013b). *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

Willhalm, T., Popovici, N., Boshmaf, Y., Plattner, H., Zeier, A., and Schaffner, J. (2009). Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *VLDB*, 2(1):385–394.

Xilinx (2017). *Zynq UltraScale+ MPSoC Data Sheet: Overview*.

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

Zhou, J. and Ross, K. A. (2002). Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156.

Ziener, D., Bauer, F., Becher, A., Dennl, C., Meyer-Wegener, K., Schürfeld, U., Teich, J., Vogt, J.-S., and Weber, H. (2016). Fpga-based dynamically reconfigurable sql query processing. *ACM Trans. Reconfigurable Technol. Syst.*, 9(4):25:1–25:24.

Zukowski, M., Héman, S., Nes, N., and Boncz, P. A. (2006). Super-scalar RAM-CPU cache compression. In *ICDE*, page 59.