Column Scan Acceleration in Hybrid CPU-FPGA Systems

Nusrat Jahan Lisa, Annett Ungethüm,
Dirk Habich, Wolfgang Lehner
Technische Universität Dresden
Database Systems Group
Dresden, Germany
{firstname.lastname}@tu-dresden.de

Nguyen Duy Anh Tuan, Akash Kumar
Technische Universität Dresden
Processor Design Group
Dresden, Germany
{firstname.lastname}@tu-dresden.de

ABSTRACT

Nowadays, in-memory column store database systems are state-of-the-art for analytical workloads. In these column stores, a full column scan is a fundamental key operation and thus, the optimization of this primitive is very crucial from a performance perspective. For this optimization, advances in hardware are always an interesting opportunity, but represent also a major challenge. At the moment, hardware systems are more and more changing from homogeneous CPU systems towards hybrid systems with different computing units. Based on that, we focus on column scan acceleration for hybrid hardware systems incorporating a Field Programmable Gate Array (FPGA) and a CPU into a single system in this paper. The advantage of those hybrid systems is that the FPGA has usually direct access to the main memory of the CPU avoiding data copy which is a necessary procedure in other hybrid systems like CPU-GPU architectures. Thus, we present several FPGA designs for a recent column scan technique to fully offload the scan operation to the FPGA. In detail, we present our basic FPGA design and different optimization techniques. Then, we present selective results of our exhaustive evaluation showing the benefit of our FPGA acceleration. As we are going to show, we achieve a maximum speedup of factor 7 compared to a single-threaded CPU scan execution.

1. INTRODUCTION

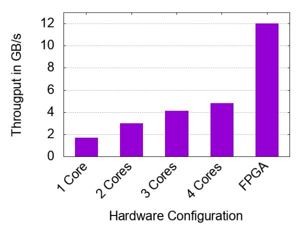
In our data-driven world, efficient query processing is still an important aspect due to the ever-growing amount of data. In fact, the growth of data even outnumbers Moore's law of digital circuit complexity [35]. Therefore, the architecture of database systems is constantly evolving, especially by adapting novel hardware features to satisfy response times and throughput demands [6, 20, 25, 30, 33]. For instance, the database architecture shifted from a disk-oriented to a main memory-oriented architecture to efficiently exploit the ever-increasing capacities of main memory [1, 22, 27, 37]. This in-memory database architecture is now state-of-the-art and characterized by the fact, that all relevant data is completely stored and processed in main memory. Additionally, relational tables are organized by column rather than by row [1, 6, 8, 22, 37] and the traditional tuple-at-a-time query processing model was replaced by newer and adapted processing models like column-at-a-time or vector-at-a-time [1, 6, 22, 37, 48].

To further increase the performance of queries, in particular for analytical queries in these in-memory column stores, two key aspects play an important role. On the one hand, data compression is used to tackle the continuously increasing gap between computing power of CPUs and memory bandwidth (also known as memory wall [6]) [2, 5, 9, 21, 47]. Aside from reducing the amount of data, compressed data offers several advantages such as less time spent on load and store instructions, a better utilization of the cache hierarchy, and less misses in the translation lookaside buffer. On the other hand, in-memory column stores constantly adapt to novel hardware features like vectorization using Single-Instruction Multiple Data (SIMD) extensions [34, 48], GPUs [20, 26] or non-volatile main memory [33].

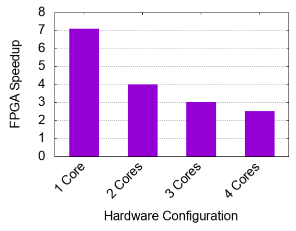
From a hardware perspective, we currently observe a shift from homogeneous CPU systems towards hybrid systems with different computing units mainly to overcome physical limits of homogeneous systems [11, 29]. In particular, hybrid hardware systems incorporating a Field Programmable Gate Array (FPGA) and a CPU are emerging, being very interesting from a performance perspective. Generally, FPGAs are integrated circuits, which are configurable after being manufactured. Thus, FPGAs can be used as a hardware extension to the database system where some specialized functionality is efficiently implemented. Additionally, FPGAs have usually direct access to the main memory of the CPU in such hybrid systems. In contrast to other hybrid systems like CPU/GPUs, this direct main memory access is unique regarding to avoid the bottleneck of copying data between the different computing units [14, 26].

Our Contribution

A core primitive in in-memory column stores is a *column scan* [12, 31, 40], because analytical queries usually compute aggregations over full or large parts of columns. Thus, the optimization of this scan primitive is very crucial from a performance perspective and several software-based approaches have been proposed [12, 31, 40]. Some of these approaches are already tailored to hardware features like SIMD vectorization as optimization [12, 40]. Generally, the task of a column scan is to compare each entry of a given



(a) Scan Througput



(b) FPGA Speedup

Figure 1: Comparing Scan Throughout for CPU as well as FPGA execution. More details can be found in Section 4.

column against a given predicate and to return all matching entries. All recent column scan approaches have in common that (i) they directly work on compressed data and (ii) they process multiple compressed column values within a single instruction [12, 31, 40]. As shown in [12, 31, 40], the more column values are packed together in processing registers, the more column values can be processed at once in parallel leading to better performance. However, the processing registers on common CPUs are limited to 64-bit [31] for processor words or to the size of the available vector registers [12, 40]. Generally, vector—Single Instruction Multiple Data (SIMD)—extensions such as Intel's SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions) have been available in modern processors for several years. On Intel systems, there are currently three vector sizes available: (i) 128-bit for SSE, (ii) 256-bit for AVX or AVX2, and (iii) 512-bit for AVX-512.

However, the limited sizes of the processing registers restrict the scan processing capabilities, because only a limited number of compressed column values can be processed in parallel at once. To overcome that, the utilization of FPGAs in hybrid hardware systems is a very interesting optimization candidate. To investigate this optimization candidate in a systematical way, we decided to create and to compare various FPGA designs for a recent column scan technique called *BitWeaving* [31]. Our underlying hybrid hardware foundation is a Xilinx® Zynq UltraScale+^{TM}board featuring four ARM cores with a maximum frequency of 1.2GHz and an integrated FPGA [43]. The ARM cores as well as the FPGA have direct access the main memory [43]. Figure 1(a) shows the BitWeaving scan throughput—how many bits are processed in a second—measured in GB/s for single-threaded as well as for multi-threaded execution on the ARM cores[1]. As we can see in Figure 1(a), the scan throughput improves with an increasing number of cores as expected, whereby the throughput almost saturates with four cores. Nevertheless, we achieve a much higher scan throughput with our best-performing FPGA design on the same data, whereby this FPGA design runs only at 250MHz, but processes more compressed column values in parallel. Figure 1(b) depicts the resulting FPGA speedups compared to different CPU configurations. Compared to the single-threaded execution, we achieve a speedup of seven, while the FPGA speedup to four ARM cores is still greater than 2.

[1]The CPU experiments were done with the original BitWeaving source code, which was kindly provided to us by Jignesh M. Patel. The multi-threaded execution was done using OpenMP, whereas each thread processed the same amount of data (uniform distribution).

To summarize, we make the following contributions in this paper:

1. We describe our target hybrid CPU-FPGA hardware system as well as our used example scan technique BitWeaving in Section 2.
2. In Section 3, we present our pipeline-based basic hardware design for an FPGA-accelerated BitWeaving scan. Based on that basic design, we introduce different optimization techniques in a systematical way to increase the scan throughput. This section already includes some initial evaluation results to validate our optimization techniques.
3. A more comprehensive evaluation is described in Section 4. In particular, we also compare our FPGA optimization with different SIMD optimizations. As we are going to show, the FPGA optimization is more beneficial than SIMD.

Finally, we close the paper with related work in Section 5 and a summary including future work in Section 6.

2. PRELIMINARIES

Before we systematically describe different FPGA designs for the scan acceleration, we have to introduce necessary basics in this section. The basics include two points: (i) a description of our target hybrid CPU-FPGA hardware and (ii) a brief summary of the BitWeaving scan technique [31].

2.1 Target Hybrid CPU-FPGA System

Fundamentally, Field Programmable Gate Arrays (FPGAs) are integrated circuits, which are configurable after being manufactured at any time. Internally, FPGAs are composed of programmable logic blocks, a collection of small on-chip memories, and arithmetic units (DSPs). To create a custom hardware module for an FPGA, a hardware description language, e.g., Verilog, is used to describe the operation mode of a specific application logic. This description is then translated via several steps to an implementation for the FPGA. Then, this implementation behaves like an application-specific integrated circuit (ASIC) [39]. A core component of this implementation is the state of the programmable LookUp-Tables (LUTs). These LUTs are used for implementing simple n-ary functions. More complex functions are realized by using an array of LUTs enabling the construction of custom hardware modules for any type of application logic. Typically, this offers a higher performance while maintaining a lower power dissipation than on CPUs. To be competitive with common CPUs, the custom hardware module for a specific application logic has to be well-designed, since current FPGAs usually run at very low clock-rates around $200 - 400$ MHz. To enable that, the most challenging issue is to create efficient processing pipelines due to the close proximity of logic and memories [39].

The FPGA acceleration of database operations is becoming increasingly important since hardware vendors like Intel® or Xilinx® incorporate an FPGA and a CPU into a common system. In this paper, we use a hybrid CPU-FPGA hardware system from Xilinx® called Zynq UltraScale+^{TM}as foundation [43]. The architecture of this hybrid system is depicted in Figure 2. As shown in this figure, our target system contains two major top-level blocks: the processing system (PS) and the programmable logic (PL). All programmable custom hardware FPGA modules are implemented inside the PL, whereby the maximum frequency can

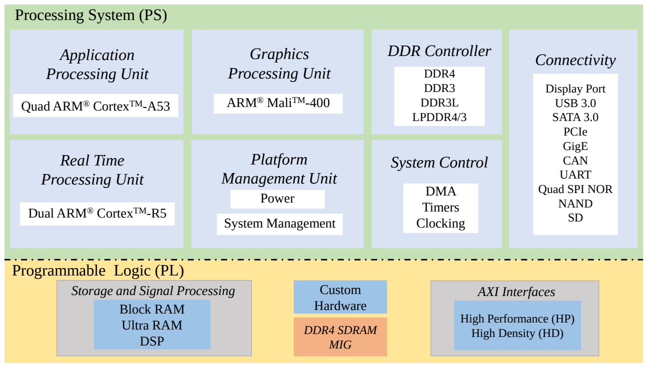


Figure 2: Zynq UltraScale+$^{\text{TM}}$hybrid architecture.

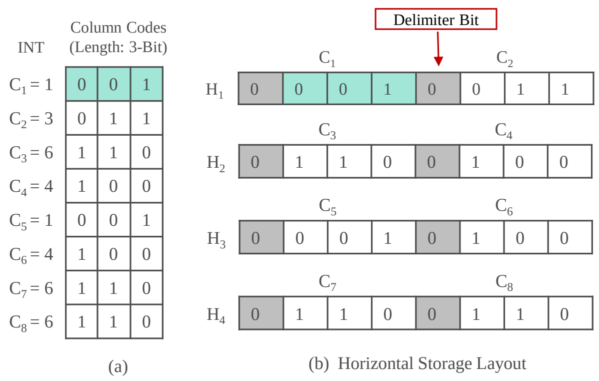


Figure 3: Illustration of BitWeaving/H layout.

be 400MHz. Inside the PS, there is a common 64-bit quad ARM® Cortex-A53 with dedicated and shared cache memories, static dual port RAM, registers, and controllers. The maximum frequency of these ARM cores is 1.2GHz. Additionally, the PS part features two ARM® Cortex-R5 cores for real time processing and a Mali-400 GPU, but both are not considered further in this paper. Moreover, the Zynq UltraScale+$^{\text{TM}}$ has two DDR4 memories. While one is located in the Processing System (PS) (4GB in size), the other is located in the Programmable Logic (PL) (512MB). In contrast to previous Xilinx® hybrid systems, the connection between the PS and the PL on the Zynq UltraScale+$^{\text{TM}}$is more powerful. Concretely, it has several High Performance AXI interfaces between PL and PS providing a data bus width of 32-bit/64-bit/128-bit. Based on that, the custom hardware modules on the PL part have direct access to the large main memory on the PS part, so that PS and PL can work on the same data elements.

2.2 Investigated Column Scan Operation

With the increasing demand for in-memory data processing, there is a critical need for fast scan operations in column store systems [12, 31, 40]. The *BitWeaving* approach addresses this need by packing multiple compressed column values into processor words and applying full-word instructions for predicate evaluations using a well-defined arithmetic framework [31]. The core idea of this technique was already published in 1975 by Lamport [28] and has been recently extended in [31]. In this extension, BitWeaving comes with two storage layout variants, a horizontal layout, *BitWeaving/H*, and a vertical layout, *BitWeaving/V*.

In the remainder of this paper, we restrict ourselves to BitWeaving/H. There are several reasons why we decided to use BitWeaving and in particular BitWeaving/H. On the one hand, BitWeaving in general is a simple and hardware-independent approach, but a very efficient technique at the same time. In contrast to that, ByteSlice and other scan approaches are tailored to specific hardware environments like vectorization and the speedups compared to BitWeaving are marginal [12, 40]. On the other hand, BitWeaving/H is currently the storage layout from a compression point of view, because many compression algorithms used in column stores are designed for a horizontal layout [9]. To convert horizontal compressed column codes to a vertical BitWeaving layout requires additional work, which would be an interesting topic for future work.

2.2.1 *BitWeaving/H Storage Layout*

In-memory column stores have more or less a common storage approach: (i) encode values of each column as a sequence of integers using some kind of dictionary coding [2, 5] and (ii) apply lightweight lossless data compression to each sequence of integers resulting in a sequence of compressed column codes [1, 2, 21]. An example is shown in Figure 3(a), where eight 32-bit integer values C_i are represented using 3-bit compressed column codes. Fundamentally, BitWeaving assumes a fixed-length order preserving compression scheme, so that all compressed column codes of a column have the same bit length [31]. Then, the bits of the column codes are aligned in main memory in a way that enables the exploitation of intra-cycle parallelism using ordinary processor words. As illustrated in Figure 3(b), the column codes are contiguously stored in processor word H_i in BitWeaving/H, where the most significant bit of every code is used as a delimiter bit between adjacent column codes. In our example, we use 8-bit processor words, so that two 3-bit column codes fit into one processor word including one delimiter bit per code. The delimiter bits are used later to store the result of a predicate evaluation.

2.2.2 *BitWeaving/H Predicate Evaluation*

The task of a column scan is to compare each column code with a constant C and to output a bit vector indicating whether or not the corresponding code satisfies the comparison condition. To efficiently perform such a column scan using the BitWeaving/H storage layout, Li et al. [31] proposed an arithmetic framework to directly execute predicate evaluations on the compressed column codes. There are two main advantages: (i) predicate evaluation is done without decompression and (ii) multiple column codes are simultaneously processed within a single processor word using full-word instructions (intra-cycle parallelism) [31]. The supported predicate evaluations include equality, inequality, and range checks, whereby for each evaluation a function consisting of arithmetical and logical operations is defined [31].

Figure 4 highlights the *equality* check in an exemplary way, whereby the other predicate evaluations work in a similar way. The input from Figure 3(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

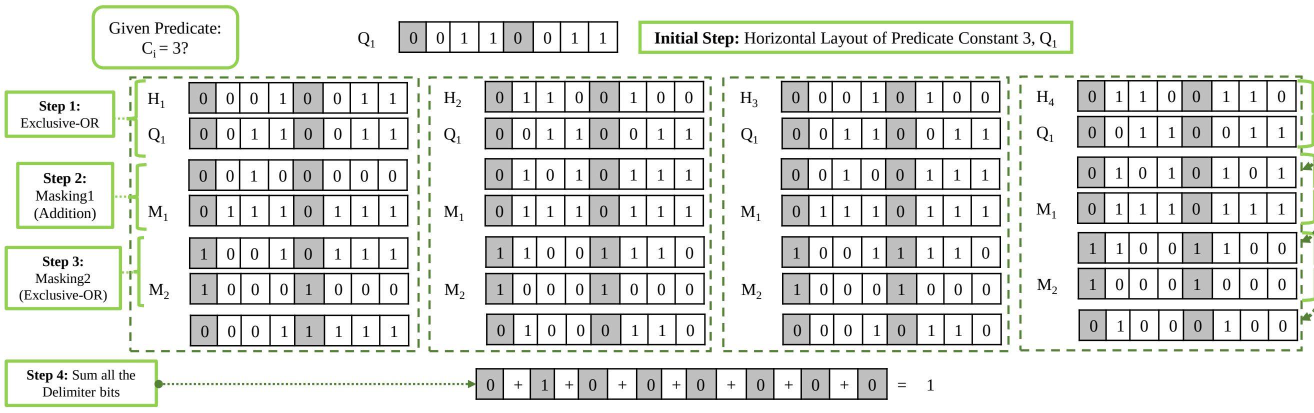

Given Predicate: C_i = 3?
Q_1
Initial Step: Horizontal Layout of Predicate Constant 3, Q_1
Step 1: Exclusive-OR
Step 2: Masking1 (Addition)
Step 3: Masking2 (Exclusive-OR)
Step 4: Sum all the Delimiter bits
H_1 H_2 H_3 H_4
M_1 M_2
0 + 1 + 0 + 0 + 0 + 0 + 0 + 0 = 1

Figure 4: Equality predicate evaluation with the BitWeaving/H technique.

number *3* are converted into the BitWeaving/H storage layout (H_1, H_2, H_3, H_4) and Q_1, respectively.

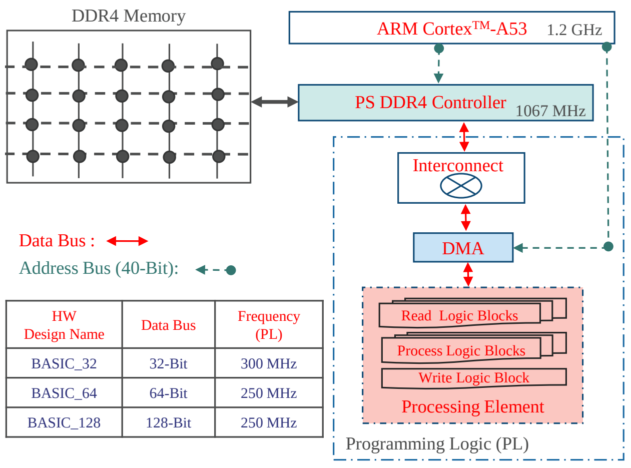
Step 1: An *Exclusive-OR* operation between the words (H_1, H_2, H_3, H_4) and Q_1 is performed.

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 are set to one and the rest of all bits is set to zero) is performed.

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

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



HW Design Name	Data Bus	Frequency (PL)
BASIC_32	32-Bit	300 MHz
BASIC_64	64-Bit	250 MHz
BASIC_128	128-Bit	250 MHz

Figure 5: Pipeline-based basic design for *BitWeaving/H* scan using one data channel.

3. FPGA-BASED SCAN ACCELERATION

In this section, we present different FPGA designs for the BitWeaving scan acceleration in a systematical way. We start with pipeline-based basic designs, followed by two optimization approaches. The first approach tries to read more data from main memory using multiple data channels, while the second optimization uses multiple memory locations. In each part, we also present some initial evaluation results to guide the reader through the different design alternatives. More evaluation results will be discussed in Section 4.

3.1 Pipeline-based Basic Designs

Custom-made hardware (we also call it *Processing Element (PE)*) is implemented inside the PL using Configurable Logic Block (CLB) slices [39]. CLB slices contain LUTs (LookUp Tables), Flip-Flops (FF), Arithmetic Carry Logic and Multiplexers [32] [39]. As described earlier, the *BitWeaving/H* scan technique is based on an arithmetic framework containing only arithmetic as well as logical operations [31]. Thus, this scan should and can be easily realized on FPGAs. However, the biggest strength of FPGAs lies in their inherent hardware parallelism, which is also the most challenging issue [39].

Pipeline parallelism is a very attractive parallelism opportunity for FPGAs due to the close proximity of logic and memories [39]. To realize pipeline parallelism, a set of data processing blocks has to be connected in series, whereby the output of one block is the input of the next one. Then, the blocks are executed in parallel. To enable that for *BitWeaving*, we designed a specific *Processing Element* for the *BitWeaving/H* scan consisting of three different blocks as shown in Figure 5, whereby we stream data from the large PS main memory through our processing element in the PL.

Read Logic Block: Inside our processing element, there are two read blocks necessary, one is for reading the predicate constant word and the other is for reading the raw data words (the words containing a number of column codes, which are checked for equality).

Process Logic Block: This block is responsible for the predicate evaluation. Based on the type of predicate, there can be more than one process block in parallel: For example, four operations are required to perform an equality check: (i) *Exclusive-OR*, (ii) *Masking1*, (iii) *Masking2*, iv) *Preparing* the output (see Figure 4). In this case, there are four process blocks which are processing more than one data word in parallel.

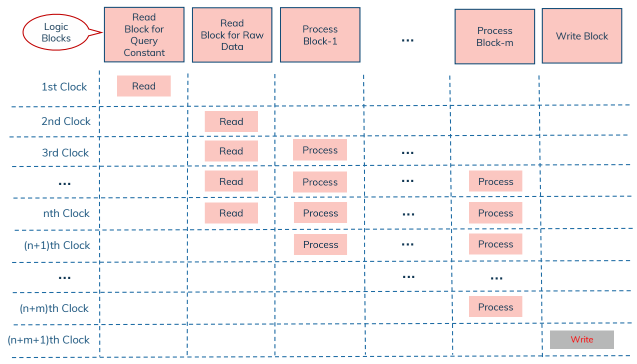


Figure 6: Timing diagram of our pipeline-based basic FPGA design for *BitWeaving/H* scan.

Write Logic Block: In the write block, the final result of the predicate evaluation is written back to PS main memory.

As illustrated in Figure 5, our *BitWeaving/H scan processing element* is connected to the main memory of the PS part through a single data channel using a DMA (direct memory access) controller (High Performance AXI interfaces), so that we can directly stream data words to the *BitWeaving/H* processing element. Based on that design, each block processes one data word in each clock cycle, while in each clock cycle more than one block is executed in parallel as shown in Figure 6. For example, if there are m process blocks to process n words, then it takes *(m+n+1)* clock cycles (see Figure 6), whereas a non pipeline-based design would take *(mn+1)* clock cycles. Therefore, each data word contains multiple compressed column codes and in each cycle multiple data words are simultaneously processed offering a high degree of parallelism.

The data word size on our target system can be configured as 32-bit, 64-bit or 128-bit, because the interfaces between the PL and PS provide a data bus width of corresponding sizes. For each possible bus data width, we created an appropriate design named BASIC_32, BASIC_64, BASIC_128. That means, if we read 32-bit data from the main memory, we also process 32-bit in our processing element. The same applies for 64-bit and 128-bit.

Interim Evaluation. To show the applicability and effect of our basic hardware design, we present some initial evaluation results. For the initial evaluation in this section, we generated a data set containing 1 million column codes, whereby each column had a fixed size of 3-bit. This data set is stored in the large main memory of the PS part. For comparison, BitWeaving using the original source code was executed on a single ARM core and we measured a throughput of 1.9 GB/s as illustrated in Table 1. This execution used a processor word size of 64-bit, so that sixteen 3-bit column codes are processed simultaneously. During the scan, an equality check had been conducted.

Table 1 also shows the throughput results for our different basic designs. As we can see, the throughput increases with increasing data widths as expected. The reason is that the larger the data width, the higher the parallelism leading to a higher throughput. However, only the 32-bit basic design can be executed with a frequency of 300MHz. For the other designs, we had to drop the frequency to 250MHz to guarantee a reliable and valid processing. Therefore, the speedup of the 64-bit design compared to the 32-bit design cannot be two. Nevertheless, the speedup of BASIC_128 compared to BASIC_64 is two, because the frequency stays constant and twice the amount of data is processed in a single clock.

Design	Throughput	Frequency
1 ARM Core	1.9GB/s	1.2GHz
BASIC_32	1.1GB/s	300MHz
BASIC_64	1.9GB/s	250MHz
BASIC_128	3.9GB/s	250MHz

Table 1: Initial evaluation results for our basic designs compared to a single-threaded execution on an ARM core.

Interim Conclusion. Compared to a single-threaded CPU execution running on 1.2GHz, our BASIC_64 FPGA design at 250MHz delivers the same throughput. With a data width of 128, we can even double the throughput. Based on that, we are able to conclude that (i) the FPGA scan acceleration is possible, (ii) higher data widths are more beneficial even if the frequency gets lower, and (iii) using the full possible data width of the data channel delivers the best performance.

3.2 Optimization using Multiple DMAs

To further improve the scan throughput, we have to process more data in parallel. Unfortunately, the data bus width of the PS DDR4 controller is limited to 128-bit per data channel. But inside the PL, we can instantiate up to 1024-bit data bus width based DMAs and inside our processing elements we can increase the processing word size as well. To efficiently utilize these PL properties, we have to go for data parallelism using multiple data channels on the PS part, whereas four non-coherent channels are available on out target hybrid system. Based on that, there are two processing opportunities: (i) combined processing and (ii) independent processing. Both as well as a hybrid approach are presented and discussed subsequently.

3.2.1 Combined Processing

To parallelize the scan of data beyond 128-bit words in a single cycle, we created designs using multiple DMAs (data channels) as shown in Figure 7. Our pipeline-based basic design still serves as foundation, whereby multiple 128-bit data words are combined before processing. For example, our 256-bit design uses two DMAs and the resulting two 128-bit data words are concatenated into a 256-bit data word for processing. These 256-bit data words are then processed using one single processing element working on 256-bit operations. The necessary combiner is an additional pipeline stage which integrates seamlessly into our pipeline-based design, whereby the combiner is connected with multiple DMAs. Since only four non-coherent data channels are available on PS DDR4 controller, we are able to generate designs up to a data width of 512-bit. The designs are denoted as COMBINED_256, COMBINED_384, COMBINED_512. Important to note, the necessary BitWeaving operations natively operate on the different data word sizes.

Interim Evaluation. We also evaluated these designs using our initial evaluation setup as presented above, whereby we uniformly distributed the amount of data among the used

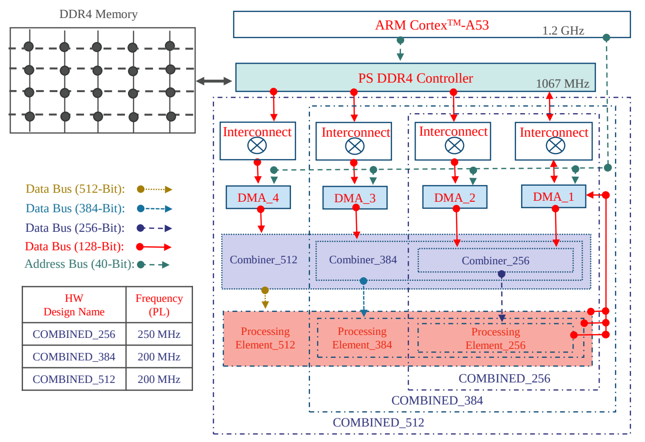


Figure 7: Combined processing design using multiple DMAs.

DMAs. The results are depicted in Table 2. As we can see, we are able to achieve a throughput of 7.7GB/s for COMBINED_256 which is almost twice as much as for the BASIC_128 design, whereby both are able to run on 250MHz. The throughput of COMBINED_384 and COMBINED_512 is less than COMBINED_256, because these designs can only run at 200MHz. There are two reasons for this behavior. First, the words, which are processed by the PEs and the Combiners are larger than in the COMBINED_256-design. This increases the number of used LUTs and Flip-Flops a signal has to pass and therefore, the time needed for a processing cycle increases. Second, each processing element is connected with more than one DMA via a combiner. However, the DDR4 controller operates multiple DMAs in a round robin manner. That means, the DDR4 controller transfers the whole burst of words to one DMA, while the others have to wait. Thus, each processing element has to wait for the other DMAs to process a single word, which has two effects: (1) The frequency is decreased leading to a drop in the scan throughput, and (2) the instructions on a PE are stalled until their DMA was provided with the necessary data, which introduces additional stalling cycles.

Interim Conclusion. From this initial evaluation, we can conclude that the combined processing approach is most beneficial for 256-bit data words by using only two data channels on our target system. In all other cases, the frequency goes down which has a negative effect on the throughput. The main reason for this behavior is that only a single FPGA processing element is interacting with multiple DMAs.

3.2.2 *Independent Processing*

In contrast to the combined approach, we also can use multiple DMAs, but process each DMA output independently by replicating our specific processing element. This approach is depicted in Figure 8. The advantages are: (i) we do not need an additional combiner and (ii) the processing is limited to 128-bit operations in the processing elements. The disadvantage is that we have to replicate the processing elements multiple times leading to a higher resource utilization. Again, we are able to use up to four non-coherent data channels leading to three designs using virtual data widths of 256, 384, and 512, but the processing is done in parallel in 128-bit physical chunks. The necessary BitWeaving operations only operate on a 128-bit data word size.

Design	Throughput	#DMAs	Frequency
BASIC_128	3.9GB/s	1	250MHz
C_256	7.7GB/s	2	250MHz
C_384	7.5GB/s	3	200MHz
C_512	6.5GB/s	4	200MHz

Table 2: Initial evaluation results for combined processing, whereby C_256 equals COMBINED_256, C_384 equals COMBINED_384, and C_512 equals COMBINED_512.

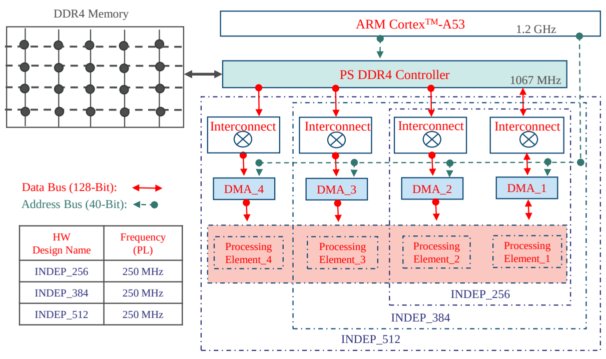


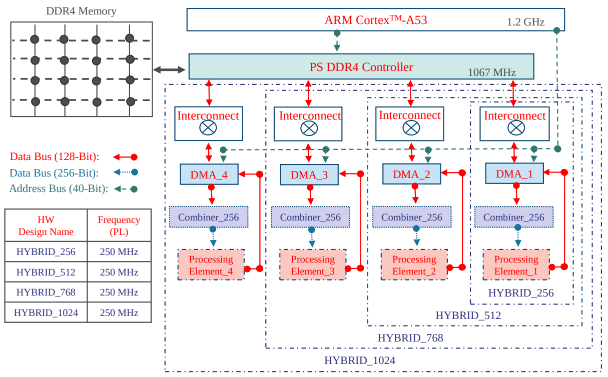
Figure 8: Independent processing using multiple DMAs.

Interim Evaluation. Table 3 shows the evaluation results for these designs using our initial evaluation setup with 1 million 3-bit column codes and with equality check during the scan. As expected, we achieve a throughput of 7.8GB/s for INDEP_256 which is almost twice the throughput of BASIC_128. In this case, COMBINED_256 and INDEP_256 show a similar behavior running both with 250MHz. However, the throughput of INDEP_384 using 3 DMAs increases compared to the combined processing approach. The throughput for INDEP_512 decreases slightly. The reasons for this are diverse. On the one hand, only three data channels are directly connected to PS DDR4 controller on our target system, the fourth is connected indirectly. On the other hand, the PS DDR4 controller theoretically runs with a frequency of 1067MHz, but physically the IO bus frequency is 949MHz including refresh cycles (measured with the Xilinx Frequency Analyzer Tool). Based on that, when all non-coherent channels are used, the controller cannot operate all channels within 949MHz which decreases the throughput. However, all independent processing designs are able to run at 250MHz.

Interim Conclusion. From a performance perspective, the independent processing approach is superior to the combined processing approach in general. The utilization of 3 data channels delivers the highest throughput so far.

Design	Throughput	#DMAs	Frequency
BASIC_128	3.9GB/s	1	250MHz
INDEP_256	7.8GB/s	2	250MHz
INDEP_384	8.7GB/s	3	250MHz
INDEP_512	8.1GB/s	4	250MHz

Table 3: Initial evaluation results for independent processing using multiple DMAs.



HW Design Name	Frequency (PL)
HYBRID_256	250 MHz
HYBRID_512	250 MHz
HYBRID_768	250 MHz
HYBRID_1024	250 MHz

Figure 9: Hybrid processing design using multiple DMAs.

3.2.3 Hybrid Processing

Nevertheless, each previous approach has its own advantages. From the combined processing, we can conclude that a 256-bit data word processing size is beneficial. For higher data word sizes, an independent processing should be preferred. In our third approach, we investigated a hybrid approach by combining both previous approaches in an appropriate way. That means, parallel processing multiple 256-bit data words in an independent way.

In our hybrid design, each processing element is connected with only one DMA via a combiner (namely Combiner_256), i.e, in COMBINED_256. In every second clock cycle, the combiner concatenates two 128-bit words into a 256-bit word and sends it to the processing element. Thus, the data bus width between processing elements and combiners is 256-bit and the rest of the design is using 128-bit data buses (see Figure 9). Then, we replicated this design 2/3/4 times as in the independent processing to realize 512/768/1024-bit (namely HYBRID_512, HYBRID_768 and HYBRID_1024) based designs, respectively in order to process more data words in parallel.

Interim Evaluation. Table 4 shows the evaluation results for these hybrid designs using column code widths of 3-bit and for equality check during the scan. With this design approach, we can slightly improve the maximum throughput compared to our pure independent processing approach, whereby each design can also run at 250MHz like the basic design for 128-bit. Interestingly, we can achieve an even better throughput for a single data channel. In this case, the throughput increases from 3.9GB/s to 4.6GB/s. Nevertheless, the design with 3 data channels delivers the highest throughput. The utilization of four channels is again not effective due to the above mentioned reasons.

3.2.4 Conclusion

To improve the scan throughput, the utilization of multiple DMAs is beneficial, whereby various approaches are

Design	Throughput	#DMAs	Frequency
BASIC_128	3.9GB/s	1	250MHz
HYBRID_256	4.6GB/s	1	250MHz
HYBRID_512	7.7GB/s	2	250MHz
HYBRID_768	9.2GB/s	3	250MHz
HYBRID_1024	8.4GB/s	4	250MHz

Table 4: Initial evaluation results for hybrid processing.

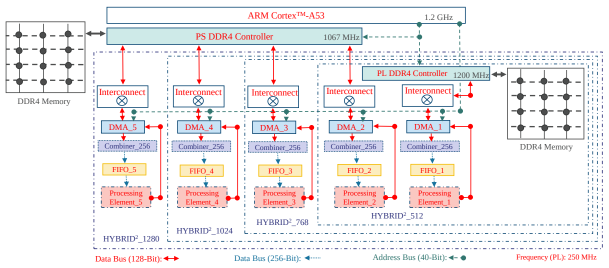


Figure 10: Hybrid^2 processing design using multiple DDR4.

possible. Each approach has its own advantages and disadvantages as presented. Nevertheless, the maximum throughput which can be achieved on our FPGA system is 9.2GB/s, whereby we only use 3 out of 4 possible data channels. The design with the highest scan throughput is HYBRID_768. The utilization of all 4 data channels always leads to a decreasing throughput compared to 3 data channels.

3.3 Optimization using Multiple Memories

Our target hybrid CPU-FPGA system has two DDR4 memories as mentioned in Section 2. To further improve the scan throughput, we also investigated the utilization of both memories (PS and PL part memory) as optimization technique, whereby we are able to easily integrate the PL part memory in all of our previous designs. The integration is done using an additional DMA controller including an additional DDR4 controller for the PL memory as illustrated in Figure 10, which allows the realization of larger data word sizes for our approaches. Figure 10 shows the extension of our hybrid memory processing optimization up to a data word size of 1280. We call this optimization HYBRID^2, because it combines our hybrid processing with hybrid (multiple) DDR4 memories.

Additionally, we extended the designs with FIFOs as temporary storage between the combiners and the processing elements. We used FIFOs because the DDR4 controller controls multiple DMAs in a round robin manner and the clock speed between DDR4 controller and PL part is different. It may happen that the DDR4 controller is sending data to a DMA while the buffer of that particular DMA is full. In that case, the DDR4 controller either has to wait for that DMA or switch to another DMA. As a consequence, the execution time will increase, which has a negative effect on the throughput. If there is an appropriate depth based FIFO, then the DMA buffer will never get full. Therefore, we prepare 2/3/4/5 DMAs based designs (namely, HYBRID^2_512, HYBRID^2_768, HYBRID^2_1024, HYBRID^2_1280, respectively) where, in every design, one DMA is connected to the PL DDR4 memory and each DMA has individual non-coherent data channels to the DDR4 controller (see Figure 10).

Interim Evaluation. Again, we initially evaluated these designs, whereby Table 5 shows the results for our initial evaluation setup. In all cases, the data is uniformly distributed among the different DDR4 memories based on the used number of DMAs. As we can see, design HYBRID^2_1024 gives the maximum throughput of 12GB/s, where three data channels (DMAs) of PS DDR4 and one data channel of PL DDR4 are used. That means, the optimization using multiple DDR4 memories is very beneficial. The throughput for

Design	Throughput	Frequency
HYBRID^2_512	7.73GB/s	250MHz
HYBRID^2_768	11.1GB/s	250MHz
HYBRID^2_1024	12GB/s	250MHz
HYBRID^2_1280	9.27GB/s	250MHz

Table 5: Evaluation results for HYBRID^2 optimization.

HYBRID^2_1280 decreases due to the utilization of four PS channels as presented above.

4. EVALUATION

In this section, we present a more comprehensive evaluation of our FPGA scan acceleration. In particular, we compare our FPGA acceleration with vectorized as well as un-vectorized CPU execution.

4.1 Comparison with ARM CPU

In Section 3, we already presented initial results of our evaluation. Figure 11 summarizes these initial results in a visual way. As introduced above, this evaluation is based on a generated data set containing 1 million column codes, whereby each column had a fixed size of 3-bit. This data set is stored in the large main memory of the PS part. To show the FPGA acceleration behavior on our target hybrid CPU-FPGA system, we also executed the BitWeaving scan on the ARM-CPUs using the original source code. This CPU execution was done using 64-bit processor words (non-vectorized). For the single-threaded CPU execution, we achieved a throughput of 1.9GB/s, while the execution on all four ARM cores, got a throughput of 4.8GB/s. The multi-threaded execution was done using OpenMP, whereby each ARM core processed the same number of column codes (uniform distribution). During the scan, an equality check had been conducted.

As we can clearly see in Figure 11, there is only one FPGA design, BASIC_32, which has a lower throughput than the single-threaded CPU execution. All other designs achieve an equal or even higher throughput. Compared to a multi-threaded CPU execution using 4 cores, most of our FPGA designs achieve a higher throughput, too. The best-performing FPGA design running only on the PS main memory (HYBRID_768) has a throughput of 9.2GB/s, which is almost twice the throughput of the best performing multi-threaded CPU approach using 4 cores. The FPGA throughput can be increased by using the main memory of the PL part. In this case, we are able to achieve a throughput of 12GB/s with the design HYBRID^2_1024. Figure 1 summarizes these results including the illustration of achieved speedups.

4.2 Different Scan Settings

We also evaluated our FPGA designs with different column codes sizes of 7 and 15 bits, with various numbers of column values and with a *less than* predicate check. For all experiments, we almost achieved the same throughput results. For example, Figure 12 shows the throughput results for a *less than* predicate check on 1 million 3-bit column codes. If we compare Figure 11 and 12, only marginal differences are visible. In all experiments, we achieved almost the same throughput with our BASIC_64 design as with the single-threaded CPU variant. Additionally, our designs using more than a single data channel deliver a higher throughput than the multi-threaded execution using four ARM cores. Nevertheless, the designs with only 3 data channels are more efficient than the other variants.

That means, offloading the scan primitive to FPGA is very beneficial. On the one hand, we achieve a higher throughput than the execution on the ARM cores. On the other hand, the ARM cores are free to execute other tasks. Moreover, it is already well-known that FPGAs are more energy-efficient than common CPUs.

4.3 Data Distribution

One of the important constraints—which we observed during our experiments—is the type of data distribution (even or non-even) among the DMAs. In particular, when an odd number of DMAs are used in the design. Multiple DMAs get access by the processor to buffer data in a round robin manner. For example, the non-even data distribution ($\frac{1}{2}$:$\frac{1}{8}$:$\frac{3}{8}$) among three DMAs takes more cycles than the even distribution of data ($\frac{1}{3}$:$\frac{1}{3}$:$\frac{1}{3}$) (see Figure 13 and Figure 14). In a round robin manner, order preserving access and the same burst ratio of each DMA introduce ideal time during non-even data distribution, which affects the clock cycles and the throughput negatively. As a result, distributing data

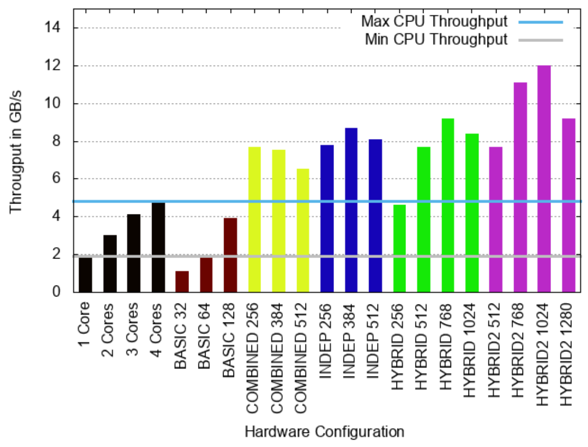


Figure 11: Throughput results for 3-bit columns codes and equality predicate check.

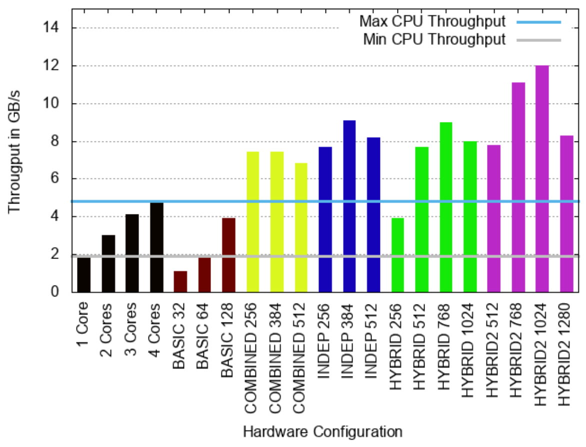
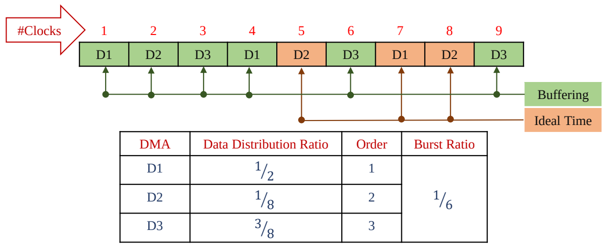
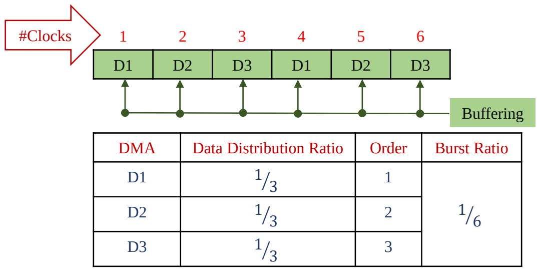


Figure 12: Throughput results for 3-bit columns codes and less than predicate check.



DMA	Data Distribution Ratio	Order	Burst Ratio
D1	$^1/_2$	1	$^1/_6$
D2	$^1/_8$	2	
D3	$^3/_8$	3	

Figure 13: Non-evenly data distribution among three DMAs.



DMA	Data Distribution Ratio	Order	Burst Ratio
D1	$^1/_3$	1	$^1/_6$
D2	$^1/_3$	2	
D3	$^1/_3$	3	

Figure 14: Evenly data distribution among three DMAs.

evenly among DMAs results in higher scan throughput in all cases. Thus, all our presented evaluation results are based on evenly distributed data.

4.4 Resource Utilization

Table 6 gives an overview on the relative resource utilization (number of LUTs and FF) of our designs for the scan with equality predicate check. As we can see, the hardware resource utilization of all designs is comparatively marginal, whereby the resource utilization increases with increasing data sizes. Interestingly, the independent processing designs are only slightly larger than the combined processing designs. Furthermore, the resource utilization of our hybrid approaches is higher, in particular for the hybrid memory optimization. That was to be expected.

4.5 Comparison with SIMD

Generally, our approach has many similarities to vectorization using SIMD extensions. To show the differences, we vectorized BitWeaving/H to be able to run on various vector register sizes like 128−, 256−, and 512−bit, which are currently available on Intel CPU Xeon systems. 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 [31]. The predicate evaluation works exactly as described in Section 2, but instead of arithmetic operators, the corresponding vector intrinsics are used to process 2, 4, or 8 64−bit values at once depending on the used SIMD extension.

The evaluation of this SIMD implementation was done on an Intel Xeon Gold 6130 with DDR4-2666 memory offering SIMD extensions with vector registers of sizes 128- (SSE 4.2), 256- (AVX2), and 512-bit (AVX-512). The base CPU frequency of this system is 2.10GHz with a maximum turbo frequency of 3.70GHz. For column codes containing 3 bits, this original un-vectorized BitWeaving scan approach with equality check achieved a throughput of 2.9GB/s on

Design	#DMAs PS + PL	LUTS (%)	FF (%)
BASIC_32	1 + 0	2.24	1.28
BASIC_64	1 + 0	2.36	1.42
BASIC_128	1 + 0	2.62	1.68
COMBINED_256	2 + 0	3.47	2.3
COMBINED_384	3 + 0	4.34	2.88
COMBINED_512	4 + 0	5.24	3.48
INDEP_256	2 + 0	3.51	2.28
INDEP_384	3 + 0	4.37	2.88
INDEP_512	4 + 0	5.37	3.51
HYBRID_256	1 + 0	2.26	1.48
HYBRID_512	2 + 0	4.16	2.9
HYBRID_768	3 + 0	6.64	4.88
HYBRID_1024	4 + 0	8.65	6.41
HYBRID^2_512	1 + 1	9.42	6.2
HYBRID^2_768	2 + 1	11.62	7.9
HYBRID^2_1024	3 + 1	13.82	9.61
HYBRID^2_1280	4 + 1	15.82	11.25

Table 6: Resource evaluation.

Extension	Word Size	Throughput	Speedup
CPU	64-bit	2.9GB/s	-
SSE 4.2	128-bit	3.3GB/s	1.14
AVX 2	256-bit	3.5GB/s	1.21
AVX-512	512-bit	2.9GB/s	1

Table 7: Vectorization evaluation results on Intel Xeon.

this systems. The results for the different vectorizations for this setting are shown in Table 7, whereby all throughput values are averaged over 10 runs. As we can see, 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 increases 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.

We also investigated different column codes sizes and different predicate evaluation leading to similar results. Based on that comprehensive evaluation, 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. This finding was unexpected because even if the speedup is not 100% when the vector size doubles, at least the 512-bit implementation should be the fastest. To find the reason for this behavior, we measured the performance of I/O-operations, which are the most expensive operations in our scenario. On the used system, the aligned and unaligned load and store operations were faster than the stream load and store operations for a small number of threads (<16). This is why we do not use stream-operations for the single-threaded test case. Additionally, the load-operators are followed by a comparison to avoid it being optimized out by the compiler. We measured the single-threaded throughput of this comparison and the store operators for all vector widths. For the comparison, the throughput increases slightly when the vector width increases from

11.5 GB/s for 128-bit to 11.9-12.2 GB/s for 256-bit and 12.7-12.8 GB/s for 512-bit. This would imply that the 512-bit implementation is indeed the best performing one, but our test scenario also includes a significant amount of store operations, which become slower the wider the vector register is. While storing 128-bit reaches a throughput of 9.5-9.9 GB, 256-bit can be stored at 8.9-9.2 GB/s, and 512-bit even go a as low as 7.4-7.5 GB/s. The slightly better load performance for 512-bit cannot make up for this decrease of the store performance.

To summarize, compared to the throughput of our BASIC_128 design, the throughput of all vectorized variants is less. Both are comparable because both work in a single-threaded fashion. From that we conclude, that the FPGA acceleration is more beneficial than the acceleration with SIMD extensions.

5. RELATED WORK

In this section, we give an overview of related work, whereby three parts are highly relevant: (i) acceleration of database primitives and (ii) specialized hardware. In addition to the description, we also link our presented FPGA scan acceleration with related work.

5.1 Acceleration of DB Primitives

The hardware acceleration of database primitives is a very active research field. In this domain, we have to distinguish between acceleration of storage and processing operations.

5.1.1 Storage Operations

A lot of work has been done on the optimization of storage operations. For instance, FPGAs have been utilized to enhance flash controllers to push down specific logic to durable storage devices [13, 23, 41]. The operations typically supported include selection, projection, group by aggregation, and some kind of sorting and joins. The main goal of these works is to avoid unnecessary data movements between the durable devices and CPU. In principle, these works are similar to our approach, but many details are different. For example, our approach accelerates the in-memory column scan by directly working on the internal main memory-optimized data structures. In contrast to this, the works of [13, 23, 41] are aligned to the data structures for durable flash devices, which are different to in-memory data structures.

To accelerate in-memory column scans, the vectorization using SIMD extensions has been considered as hardware acceleration in [12, 40]. With our presented FPGA approach, we go one step further and we clearly showed that FPGAs have advantages over vectorization in Section 4.

5.1.2 Processing Operations

In addition to storage operations, the acceleration of relational processing operations has also been considered. There are several proposals for using FPGAs for operations like aggregation, group by, pipelined arithmetic computations [7, 38, 44, 46] or relational joins [17, 45]. 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 [36]. 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.

Aside from FPGAs, the acceleration using GPUs has been considered to a large extent. For instance, Karnagel et al. [26] studied the offloading of the grouping and aggregation operator to a GPU or He et al. [19] proposed an approach to accelerate joins on coupled CPU-GPU hardware. However, the main bottleneck in using GPUs is the necessary data transfer between CPU and GPU. Thus, data placement in such heterogeneous system is an important aspect and various approach have been proposed [18, 24, 25]. Since FPGAs have direct access to the main memory, this plays no role in hybrid CPU-FPGA systems.

5.2 Specialized Hardware

Another direction of related work is the design of specialized processors [42] or extensions to the instruction set architectures (ISA) of processors [4, 3, 15, 16] that speed up database workloads. For instance, Q100 [42] proposes an ASIC (Application-Specific Integrated Circuit) solution that implements a deep pipeline, while the work in [4, 3, 15, 16] explores how CPUs could be extended with instructions well suited to the needs of databases. Even though these approaches offer better energy efficiency than accelerators built with FPGAs, the drawback is that fixed-function custom hardware is only feasible to deploy if it can support a large set of workloads for a long time. Nevertheless, it would be highly interesting to design a special scan instruction for CPUs based on our investigated FPGA designs.

Additionally, Dreseler et al. [10] investigated to offload NUMA memory accesses to the interconnect hardware from a database perspective. As they have introduced, special HARP ASICs in SGI scale-up NUMA systems are key components connecting the IRU's processors to the systemwide NUMAlink interconnect. Then, the *Global Reference Unit (GRU)* of the HARPs provides a proprietary API to offload memory operations within a NUMA architecture. In particular, the GRU facilitates functionality to asynchronously copy memory between processors and to accelerate atomic memory operations. In their paper, they clearly showed how databases on large NUMA systems can profit from offloading their distant memory loads and utilizing the GRU instructions to increase the effective throughputs. The offloading of the scan operation would be highly interesting as presented in this paper.

6. CONCLUSION AND FUTURE WORK

With the increasing demand for in-memory data processing, there is a critical need for fast scan operations in in-memory database systems. [12, 31, 40]. The *BitWeaving* approach addresses this need by packing multiple compressed columns codes into processor words and applying full-word instructions for predicate evaluations using a well-defined arithmetic framework [31]. In this paper, we have presented and evaluated various FPGA design opportunities and optimizations for this *BitWeaving scan* in a systematical way. As we have shown, each design has its own properties and we are able to accelerate the scan compared to a single-threaded and multi-threaded CPU execution. For example, the best performing HYBRID^2_1024 design does not use all available non-coherent data channels of the PS DDR4 controller, but both available main memory modules of our target hardware system. That means, the optimal design depends on the concrete hybrid system. In our ongoing work, we will investigate this aspect in detail and we want to define appropriate hardware design rules for FPGA-accelerated columns scan designs on different hybrid CPU-FPGA systems.

7. REFERENCES

[1] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

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

[3] O. Arnold, S. Haas, G. P. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel, and W. Lehner. HASHI: an application specific instruction set extension for hashing. In *ADMS*, pages 25–33, 2014.

[4] O. Arnold, S. Haas, G. P. Fettweis, B. Schlegel, T. Kissinger, and W. Lehner. An application-specific instruction set for accelerating set-oriented database primitives. In *SIGMOD*, pages 767–778, 2014.

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

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

[7] J. Casper and K. Olukotun. Hardware acceleration of database operations. In *FPGA*, pages 151–160, 2014.

[8] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

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

[10] M. Dreseler, T. Kissinger, T. Djürken, E. Lübke, M. Uflacker, D. Habich, H. Plattner, and W. Lehner. Hardware-accelerated memory operations on large-scale NUMA systems. In *ADMS*, pages 34–41, 2017.

[11] H. Esmaeilzadeh, E. R. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *IEEE Micro*, 32(3):122–134, 2012.

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

[13] P. Francisco et al. The netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011.

[14] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data transfer matters for GPU computing. In *ICPADS*, pages 275–282, 2013.

[15] S. Haas and G. P. Fettweis. Energy-efficient hash join implementations in hardware-accelerated mpsocs. In *ADMS*, pages 26–33, 2017.

[16] S. Haas, T. Karnagel, O. Arnold, E. Laux, B. Schlegel, G. P. Fettweis, and W. Lehner. Hw/sw-database-codesign for compressed bitmap index processing. In *ASAP*, pages 50–57, 2016.

[17] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. W. Asaad, and B. Iyer. Accelerating join operation for relational databases with fpgas. In *FCC*, pages 17–20, 2013.

[18] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, 2009.

[19] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB*, 6(10):889–900, 2013.

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

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

[22] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[23] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. G. Lee, and J. Jeong. Yoursql: A high-performance database system leveraging in-storage computing. *PVLDB*, 9(12):924–935, Aug. 2016.

[24] T. Karnagel and D. Habich. Heterogeneous placement optimization for database query processing. *it - Information Technology*, 59(3):117, 2017.

[25] T. Karnagel, D. Habich, and W. Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *PVLDB*, 10(7):733–744, 2017.

[26] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing gpu-accelerated group-by and aggregation. In *ADMS*, pages 13–24, 2015.

[27] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. QPPT: query processing on prefix trees. In *CIDR*, 2013.

[28] L. Lamport. Multiple byte processing with full-word instructions. *Commun. ACM*, 18(8):471–475, 1975.

[29] W. Lehner, A. Ungethüm, and D. Habich. Diversity of processing units - an attempt to classify the plethora of modern processing units. *Datenbank-Spektrum*, 18(1):57–62, 2018.

[30] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and RDMA. In *SIGMOD*, pages 355–370, 2016.

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

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

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

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

[35] R. R. Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

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

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

[38] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. W. Asaad. Database analytics acceleration using fpgas. In *PACT*, pages 411–420, 2012.

[39] J. Teubner and L. Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

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

[41] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *PVLDB*, 7(11):963–974, July 2014.

[42] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: the architecture and design of a database processing unit. In *ASPLOS*, pages 255–268, 2014.

[43] Xilinx, Inc. *Zynq UltraScale+ MPSoC Data Sheet: Overview*, 2017.

[44] M. Yoshimi, R. Kudo, Y. Oge, Y. Terada, H. Irie, and T. Yoshinaga. Accelerating OLAP workload on interconnected fpgas with flash storage. In *CANDAR*, pages 440–446, 2014.

[45] M. Yoshimi, Y. Oge, and T. Yoshinaga. Pipelined parallel join and its fpga-based acceleration. *TRETS*, 10(4):28:1–28:28, 2017.

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

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

[48] M. Zukowski, M. van de Wiel, and P. A. Boncz. Vectorwise: A vectorized analytical DBMS. In *ICDE*, pages 1349–1350, 2012.