Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

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

High-Throughput BitPacking Compression

Erstveröffentlichung in / First published in:

22nd Euromicro Conference on Digital System Design (DSD). Kallithea, 28.-30.08.2019. IEEE, S. 643-646. ISBN 978-1-7281-2862-7.

DOI: http://dx.doi.org/10.1109/DSD.2019.00101

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-821925

SLUB
Wir führen Wissen.

TECHNISCHE
UNIVERSITÄT
DRESDEN

QUCOSA
Quality Content of Saxony

Final edited form was published in "Euromicro Conference on Digital System Design (DSD). Kallithea 2019", S. 643-646, ISBN 978-1-7281-2862-7
http://dx.doi.org/10.1109/DSD.2019.00101

High-Throughput BitPacking Compression

Nusrat Jahan Lisa*, Tuan D. A. Nguyen†, Dirk Habich*, Akash Kumar† and Wolfgang Lehner*
*Database Systems Group, TU Dresden, Dresden, Germany
nusrat.jahan_lisa@tu-dresden.de, dirk.habich@tu-dresden.de, wolfgang.lehner@tu-dresden.de
†Processor Design Group, TU Dresden, Dresden, Germany
tuan_duy_anh.nguyen1@tu-dresden.de, akash.kumar@tu-dresden.de

***Abstract*—To efficiently support analytical applications from a data management perspective, in-memory column store database systems are state-of-the art. In this kind of database system, lossless lightweight integer compression schemes are crucial to keep the memory storage as low as possible and to speedup query processing. In this specific compression domain, *BitPacking* is one of the most frequently applied compression scheme. However, (de)compression should not come with any additional cost during run time, but should be provided *transparently* without compromising the overall system performance. To achieve that, we focus on acceleration of *BitPacking* using Field Programmable Gate Arrays (FPGAs). Therefore, we outline several FPGA designs for *BitPacking* in this paper. As we are going to show in our evaluation, our specific designs provide the *BitPacking* compression scheme with *high-throughput*.**

***Index Terms*—in-memory database systems; lightweight compression; BitPacking; hybrid hardware systems; FPGA**

I. Introduction

Nowadays, *in-memory column store* database systems are state-of-the-art for analytical workloads [1], [5], [8], [12]. These systems pursue a main memory-centric architecture approach and assume that all relevant data (base data as well as intermediate query results) can be fully kept in the memory of a computer or of a computer network [1], [5], [8], [12]. To achieve that, these systems organize relational tables by columns rather than by rows [5]. Moreover, the values of each column are encoded as a sequence of integers using some kind of dictionary coding [4], [12]. Then, each sequence of integer is compressed using lossless lightweight integer compression algorithms [2], [15]. As we have shown in [6], [7], there is no single-best lossless lightweight integer compression available and the decision always depends on data as well as on hardware properties.

Nevertheless, *BitPacking (BP)* is one of the most frequently applied compression scheme is this domain, showing a very good—not always optimal—behavior for different data properties [6], [7], [15]. The basic idea of BP is to partition a sequence of integer values into blocks and to compress the values within each block separately by omitting leading zeros (null suppression). The number of bits used to represent every value in a block is determined by the effective bit width of the largest value in that block. Thus, BP compression consists of the following steps: (i) partition sequence of integer values into blocks, (ii) read values in each block to determine the bit width of the largest value in the block, (iii) read the values again for *bit packing* based on the largest bit width found in the previous step, and (iv) write packed words to output. To reduce the computational effort for compression and decompression, these algorithms are usually vectorized [6], [7], [15]. Thus, the block length depends on the used vector length [7], [15]. For example, for a vector length of 128-bit, the number of integers per block has to be 128 to get an aligned output of compressed values [15].

However, (de)compression is always an additional processing step which should not come with additional cost during run time. Thus, (de)compression should be provided *transparently* without compromising the overall system performance. To achieve that, advances in hardware are always offering an interesting alternative, 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. In particular, hybrid hardware systems incorporating a Field Programmable Gate Array (FPGA) and a CPU are emerging, being very interesting from a performance perspective to specialize to functionality on the FPGA. 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 the possibility to avoid the bottleneck of copying data between the different computing units [10], [14].

Our Contribution and Outline:

Based on that, we focus on *BitPacking* compression acceleration by offloading such functionality to Field Programmable Gate Arrays (FPGAs) in a hybrid hardware system setting. In detail, we make the following contributions in this paper:

1) In Section II, we briefly outline our FPGA-based implementation approach for *BitPacking* compression. We mainly focus on the compression part as a first specific step.
2) Following that, we describe our target hardware platform and present selective results of our exhaustive evaluation in Section III. As we are going to show, our specific FPGA designs achieve a very high memory throughput. Since lightweight integer compression algorithms are memory bound, we basically have focused our evaluation on examining the specified memory throughput.

Finally, we close the paper with related work in Section IV and a summary in Section V.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Euromicro Conference on Digital System Design (DSD). Kallithea 2019", S. 643-646, ISBN 978-1-7281-2862-7
http://dx.doi.org/10.1109/DSD.2019.00101

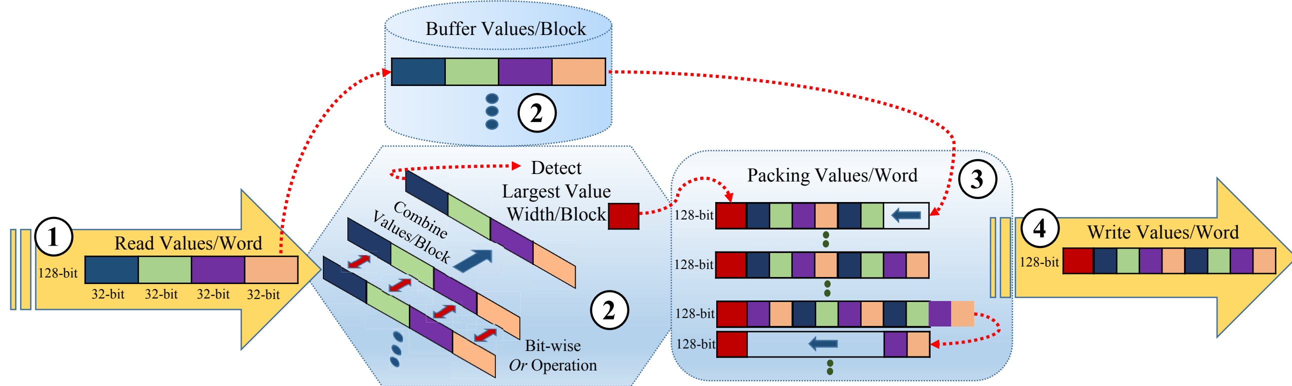

Fig. 1: Custom *BitPacking (CBP)* Overview—Flows for Offloading Values per Pipeline Stage.

II. FPGA-aware BitPacking Implementation

In this section, we briefly describe our developed FPGA-aware *BitPacking (BP)* compression implementation and different hardware design approaches incorporating multiple accelerators for the best possible compression throughput.

Generally, BP belongs to the Null Suppression (NS) algorithm class for integer compression that means it addresses the physical level of bits and bytes to reduce the number of bits per integer value by omitting leading zeros [7], [15]. For that, BP partitions a sequence of integer values into blocks and compresses each value within the block with the same bit width. This bit width is determined by the bit width of the largest value per block. Thus, two similar read operations are required—(i) for determining the bit width of the largest value in a block and (ii) during packing the values per block. Accessing main memory twice just to read the same set of values is inefficient. As a consequence, an effective FPGA option is to use internal buffers with a depth which equals the block size to temporarily store the integer values. Additionally, these buffers help filling up the pipeline stages in an FPGA-aware *BP* implementation.

Fig. 1 illustrates our pipline-based FPGA accelerator for *BP* called *CBP* (*Custom BitPacking*, whereas one DMA (Direct Memory Access) is able to access the main memory with a width of 128-bit. That means also, that our block size is 128. Our *CBP* works as follows:

① Read a 128-bit input word per clock cycle, each input word contains four 32-bit integer values.

② Store the input word into the buffer. In parallel, to detect the bit width of largest value, perform bit-wise *OR*-operations between input words per block to create a combined word. Afterwards, determine the width of the combined word by finding the left-most bit of which value is 1. This operation is done by using predefined mask registers to achieve a constant one clock cycle latency.

③ Packing buffer values into the output words, while each buffer value is compressed with largest value bit width per block by performing bit-wise *Right-Shift* operation. During packing, the most significant 8-bits of each 128-bit output word contains the bit width of the largest value per block and the remaining 120-bit are used for packed values.

④ Write 128-bit output word, while one output word is fully packed with compressed values.

In our *CBP*, the number of values packed per output word in a block depends on the largest value bit width per block. During value packing, a misalignment problem may occur when the number of values packed per output word is not divisible by 4 as each buffer word contains 4 values. Thus, in order to avoid this misalignment, we categorized the number of values packed per output word in two parts: (i) `Div4`—the number of values packing per output word is divisible by 4, (ii) `Div2`—the number of values packing per output word is divisible by 2. As a result, we rearranged the number of values packed per output word for those cases which are not divisible by 4. It is done by assigning the nearest number which is either divisible by 4 nor by 2. For example, the number of values packed per output word for the largest value bit width of 7 is $\lfloor \frac{120}{7} \rfloor = 17$, which is neither divisible by 4 or 2, and the nearest number which is divisible by 4 or 2 is 16. Thus, the new number of values packed per output word for this example is 16. During packing values, the left over values per buffer word are packed in the following output word (see ③ in Fig. 1).

Based on this principle, we started with the development of a simple straightforward single-data-channel based hardware approach, where only one *CBP* is instantiated. We call this `Approach_1` as illustrated in Fig. 2. In this approach, we use one *DMA* module (Direct Memory Access) between main memory and our *CBP*, in order to reduce the load of the processor and to reduce the latency of accessing main memory. Afterwards, we developed multiple *DMAs*-based hardware approaches called `Approach_2`, `Approach_3`, `Approach_4`, where each *DMA* is accessing main memory via an independent data channel (see Fig. 2). As a consequence, we replicated our *CBP* and *DMA* up to 4 times as the number of available main memory data channels on our target system is 4.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Euromicro Conference on Digital System Design (DSD). Kallithea 2019", S. 643-646, ISBN 978-1-7281-2862-7
http://dx.doi.org/10.1109/DSD.2019.00101

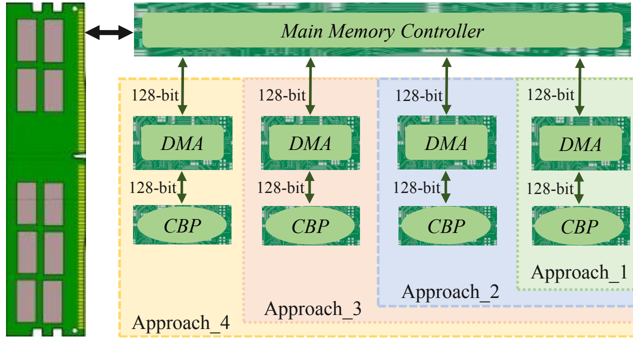

Fig. 2: Different Types of Hardware Approaches.

TABLE I: Different Categories of Data Sets.

Data Set	Data Properties
D0	uniform distribution with same value bit-width based data
D1	uniform distribution with 20% (2 to 15) bits and 80% (16 to 30) bits value-width based data
D2	uniform distribution with 50% (2 to 15) bits and 50% (16 to 30) bits value-width based data
D3	uniform distribution with 80% (2 to 15) bits and 20% (16 to 30) bits value-width based data

III. Evaluations

In this section, we experimentally analyze the behavior regarding *memory throughput* between Approach_1, Approach_2, Approach_3 and Approach_4 for different categories of data sets as *BitPacking* compression is data distribution dependent [6], [7]. Therefore, we generated 4 categories of data sets as described in Table I.

Our target system—Xilinx® Zynq UltraScale+™ —is a hybrid system containing (i) a traditional FPGA within the Programmable Logic (PL) region and (ii) an MPSoC within the Processing System (PS) region. For this paper, in PS region we only considered the 64-bit quad ARM® Cortex-A53 cores of 1.5GHz frequency and DDR4-2666 main-memory with the capacity of 4GB [19]. In addition, this system has four dedicated high performance AXI interfaces to access the main memory directly from the PL region, whereas we consider these interfaces as data-channels in our respective hardware approaches.

We start our evaluation with a *D0* category data set for 2 to 30 value bit width based data on Approach_1, Approach_2, Approach_3 and Approach_4 as illustrated in Fig.3. Approach_1 gives a symmetric throughput of 3.8GB/s (for the value bit width of 2 to 15) and 1.9GB/s (for the value bit width of 16 to 30), whereas these are the read or read-write throughput without compression for *CBP*, respectively. It defines two points:

1) our proposed implementation of *CBP* is as fast as possible, that means the latency only depends on read/write-operations not on compressing the values
2) as the value bit width increases, the compression ratio decreases and after value-width 15-bit the compression ratio is approx. 1 (see the green curve in Fig.3)

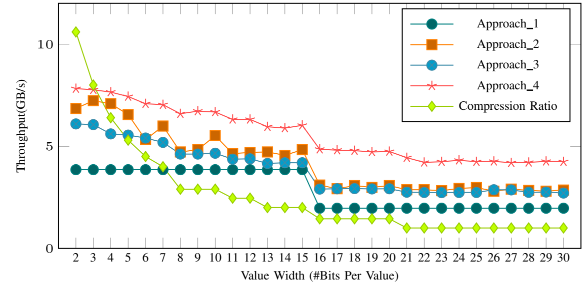

Fig. 3: Throughput Analysis on Different Hardware Approaches for a *D0* Data Set.

This defines, after value bit width of 15, there is a marginal compression happening as both input and output words may contain the same number of values. Our other hardware approaches achieve improved throughput compared to Approach_1 as the values are evenly distributed among the multiple *CBPs* for parallel compression which increase the throughput, except Approach_3 which provides mostly the same throughput as Approach_2. The reason for that is, *BitPacking* compresses values per block basis and the block size is always even and divisible by the power of 2, i.e., 128, 256, 512, etc. Thus, in Approach_3 data for compression is not evenly distributed due to block size resulting in unfavorable parallelism which affects the memory throughput. However, the throughput behavior in these other approaches per value bit-width wise is not symmetrical as Approach_1. The reason is—multiple *DMAs* interact with main memory in round robin manner—while one *DMA* is ready to interact, main memory may be busy with others, which affects the throughput [16].

Afterward, we evaluate all our hardware approaches for the *D1*, *D2* and *D3* data set. As illustrated in Fig.4, throughput in all approaches gives the maximum for *D3*, minimum for *D1* and *D2* gives in between as expected. The reason is the value bit width based data distribution as described in Table I. In all cases, Approach_4 is the winner among the others as it gives maximum throughput. However, Approach_4 utilized maximum resources (17.52% LUTs and 7.10% Flip-Flops) of FPGAs compared to other approaches. But still, the resource utilization on Approach_4 is below 25%, which is affordable.

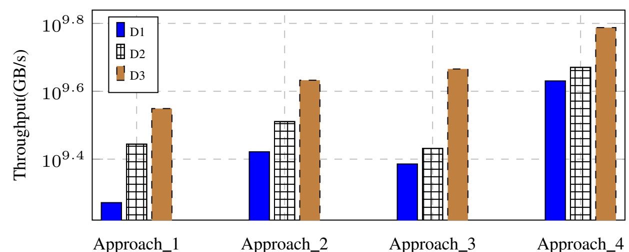

Fig. 4: Throughput Analysis for *D1*, *D2*, *D3* on Different Hardware Approaches.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "Euromicro Conference on Digital System Design (DSD). Kallithea 2019", S. 643-646, ISBN 978-1-7281-2862-7
http://dx.doi.org/10.1109/DSD.2019.00101

Finally, from all our experimental evaluations, it proves that proper pipeline-based custom *BitPacking* implementation on FPGAs is advantageous in many ways—throughput-wise and resource-wise as well. Thus, FPGA implementation should be well-investigated for other database compression techniques in the near future.

IV. Related Works

The main memory-based lightweight compression algorithms in column-store database systems are an active research field [2], [13], [20]. However, mostly the research regarding lightweight compression algorithms only considers an efficient implementation for CPU [18], [20]. In particular, CPU-intensive SIMD instruction based *BitPacking* lightweight compression algorithm implementation provides increased performance of especially analytical database queries [2], [6], [11]. Most research in the direction of FPGAs-based compression implementation focused on heavyweight compression algorithms [3], [9], [17]. For example, *Rigler et al.* presented concepts and hardware implementations using VHDL for Lempel-Ziv encoders and dynamic Huffman encoders which is suitable for the implementation of GZIP [17]. Additionally, their implementation is capable of generating compressed files that may be decompressed using a standard implementation of GZIP [17]. Moreover, *Abdelfattah et al.* explored OpenCL-based Gzip implementation on FPGAs [3]. They showed that a high-level compiler can provide competitive performance for GZIP compression and significant productivity gains compared to traditional hardware design [3]. However, *Bartk et al.* and *Fower et al.* both implemented VHDL-based Lempel-Ziv compression on FPGAs, whereby they investigate the limitations and bottlenecks of hashing table, software pipelining overhead of the Lempel-Ziv lossless compression algorithm [9].

Generall, increasing amount of data leads database researchers to concentrate on implementation for compressed database systems, whereas main memory-based lightweight compression is more effective latency-wise than heavyweight compression. In addition, lightweight compression techniques are capable to evaluate almost all types of analytical queries directly on the compressed form of data, i.e, *BitPacking* mechanism. In [16], we already presented an FPGA-approach to efficiently conduct a filter operation directly on bit-packed compressed data. To the best of our knowledge, none of the existing works investigates the domain of FPGA-based implementation of lightweight compression algorithm or the different categories of hardware approaches on FPGAs to achieve high-throughput regarding compression.

V. Conclusion

In this paper, we have presented a brief overview of our pipeline-oriented hardware implementation for high-throughput *BitPacking* compression on FPGAs. To enable seamless pipelining, we resolve algorithmic dependencies regarding read overhead and nonalignment write by introducing internal buffering and categorized value-bit packing mechanism. Although these changes sacrifice some amount of compression ratio, they enable our implementation to scale up to approx. 7.8GB/s and 4.6GB/s throughput for same and mixed value bit-width based data sets, respectively. We prepared pipeline implementation for *BitPacking* compression in a scalable and resource-efficient way. In addition, we explored different possible hardware approaches using parallelism criteria and embrace resource-throughput trade-off relations. Finally, our custom *BitPacking* implementation achieves very high throughput and resource-efficiency on hybrid CPU-FPGAs system.

References

[1] D. Abadi et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.
[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.
[3] M. S. Abdelfattah, A. Hagiescu, and D. Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *IWOCL*, pages 4:1–4:9, 2014.
[4] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.
[5] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.
[6] 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.
[7] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3):9:1–9:46, June 2019.
[8] F. Faerber et al. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.
[9] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *FCCM*, pages 52–59, 2015.
[10] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data transfer matters for GPU computing. In *ICPADS*, pages 275–282, 2013.
[11] D. Habich, P. Damme, A. Ungethüm, and W. Lehner. Make larger vector register sizes new challenges?: Lessons learned from the area of vectorized lightweight compression algorithms. In *DBTest@SIGMOD*, pages 8:1–8:6, 2018.
[12] D. Habich, P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner. Morphstore - in-memory query processing based on morphing compressed intermediates LIVE. In *SIGMOD*, pages 1917–1920, 2019.
[13] J. Hildebrandt, D. Habich, P. Damme, and W. Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *ADMS/IMDM@VLDB*, pages 40–56, 2016.
[14] T. Karnagel, D. Habich, and W. Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *PVLDB*, 10(7):733–744, 2017.
[15] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.
[16] N. J. Lisa, A. Ungethüm, D. Habich, W. Lehner, T. D. A. Nguyen, and A. Kumar. Column scan acceleration in hybrid CPU-FPGA systems. In *ADMS@VLDB*, pages 22–33, 2018.
[17] S. Rigler, W. Bishop, and A. Kennings. Fpga-based lossless data compression using huffman and lz77 algorithms. In *2007 Canadian Conference on Electrical and Computer Engineering*, pages 1235–1238, April 2007.
[18] A. Ungethüm, J. Pietrzyk, P. Damme, D. Habich, and W. Lehner. Conflict detection-based run-length encoding - AVX-512 CD instruction set in action. In *ICDE Workshops*, pages 96–101, 2018.
[19] Xilinx, Inc. *Zynq UltraScale+ MPSoC Data Sheet: Overview*, 2017.
[20] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, pages 59–59, 2006.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden